Advanced Data Structures and Algorithm Analysis

丁尧相
浙江大学

Spring & Summer 2024
Lecture 4
2024-3-18

Outline: Heaps (I)

- Review of Binary Heaps
- Leftist Heaps
- Skew Heaps
- Amortized analysis
- Take-home messages

Outline: Heaps (I)

- Review of Binary Heaps
- Leftest Heaps
- Skew Heaps
- Amortized analysis
- Take-home messages

Job Scheduling: UNIX process priorities

```
PRI COMM
 14 /System/Library/Frameworks/CoreServices.framework/Frameworks/Metadata.framework/Versions/A/Support/mdworker
 31 -bash
 31 /Applications/iTunes.app/Contents/Resources/iTunesHelper.app/Contents/MacOS/iTunesHelper
 31 /System/Library/CoreServices/Dock.app/Contents/MacOS/Dock
 31 /System/Library/CoreServices/FileSyncAgent.app/Contents/MacOS/FileSyncAgent
 31 /System/Library/CoreServices/RemoteManagement/AppleVNCServer.bundle/Contents/MacOS/AppleVNCServer
 31 /System/Library/CoreServices/RemoteManagement/AppleVNCServer.bundle/Contents/Support/RFBRegisterMDNS
 31 /System/Library/CoreServices/RemoteManagement/AppleVNCServer.bundle/Contents/Support/VNCPrivilegeProxy
 31 /System/Library/CoreServices/Spotlight.app/Contents/MacOS/Spotlight
 31 /System/Library/CoreServices/coreservicesd
...
 31 /System/Library/PrivateFrameworks/MobileDevice.framework/Versions/A/Resources/usbmuxd
 31 /System/Library/Services/AppleSpell.service/Contents/MacOS/AppleSpell
 31 /sbin/launchd
 31 /sbin/launchd
 31 /usr/bin/ssh-agent
 31 /usr/libexec/ApplicationFirewall/socketfilterfw
 31 /usr/libexec/hidd
 31 /usr/libexec/kextd
...
 31 /usr/sbin/mDNSResponder
 31 /usr/sbin/notifyd
 31 /usr/sbin/ntpd
 31 /usr/sbin/pboard
 31 /usr/sbin/racoon
 31 /usr/sbin/securityd
 31 /usr/sbin/syslogd
 31 /usr/sbin/update
 31 autofsd
 31 login
 31 ps
 31 sort
 46 /Applications/Preview.app/Contents/MacOS/Preview
 46 /Applications/iCal.app/Contents/MacOS/iCal
 47 /Applications/Utilities/Terminal.app/Contents/MacOS/Terminal
 50 /System/Library/Frameworks/CoreServices.framework/Frameworks/Metadata.framework/Support/mds
 50 /System/Library/Frameworks/CoreServices.framework/Versions/A/Frameworks/CarbonCore.framework/Versions/A/Support/fseventsd
 62 /System/Library/CoreServices/Finder.app/Contents/MacOS/Finder
 63 /Applications/Safari.app/Contents/MacOS/Safari
 63 /Applications/iWork '08/Keynote.app/Contents/MacOS/Keynote
 63 /System/Library/CoreServices/Dock.app/Contents/Resources/DashboardClient.app/Contents/MacOS/DashboardClient
 63 /System/Library/CoreServices/SystemUIServer.app/Contents/MacOS/SystemUIServer
 63 /System/Library/CoreServices/loginwindow.app/Contents/MacOS/loginwindow
 63 /System/Library/Frameworks/ApplicationServices.framework/Frameworks/CoreGraphics.framework/Resources/WindowServer
 63 /sbin/dynamic_pager
 63 /usr/sbin/UserEventAgent
 63 /usr/sbin/coreaudiod
```

When scheduler asks "What should I run next?" it could *findmin(H)*.

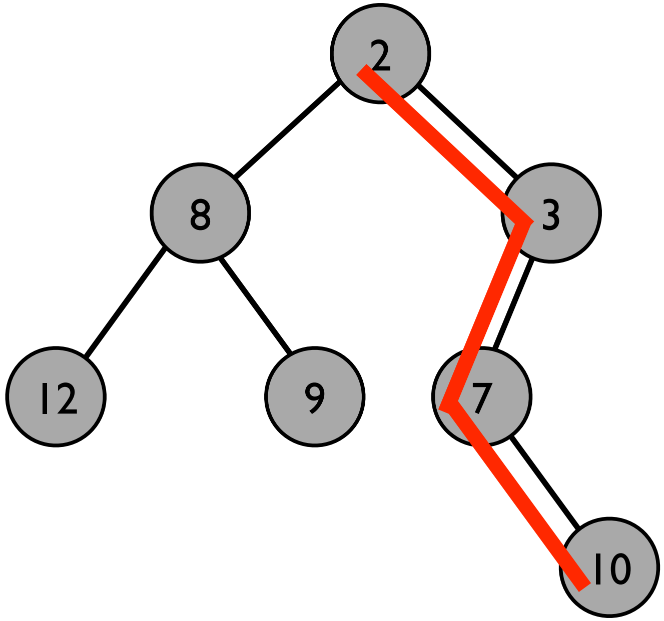
Priority Queue ADT

- Efficiently support the following operations on a set of keys:
  - *findmin*: return the smallest key
  - *deletemin*: return the smallest key & delete it
  - *insert*: add a new key to the set
  - *delete*: delete an arbitrary key

- All the balanced-tree dictionary implementations we've seen support these in O(log n) time.

- Would like to be able to do *findmin* faster (say O(1)).

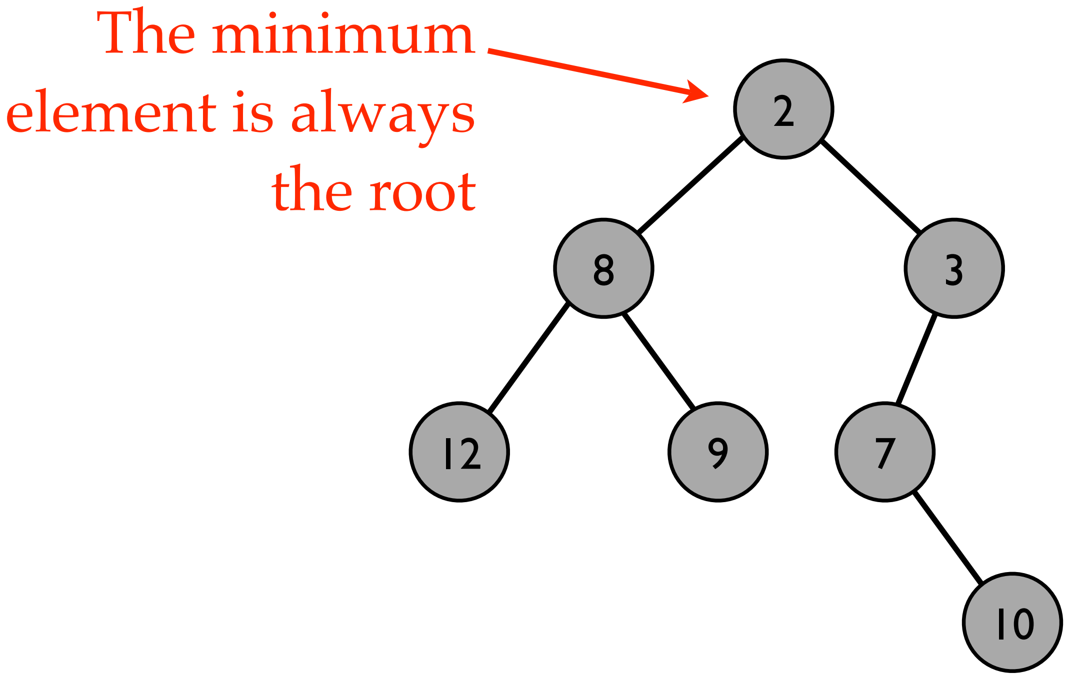
Heap-Ordered Trees



Along each path keys are monotonically non-decreasing

- The keys of the children of u are $\geq$ the key(u), for all nodes u.
- (This “heap” has nothing to do with the “heap” part of computer memory.)
- [Symmetric max-ordered version where keys are monotonically non-increasing]

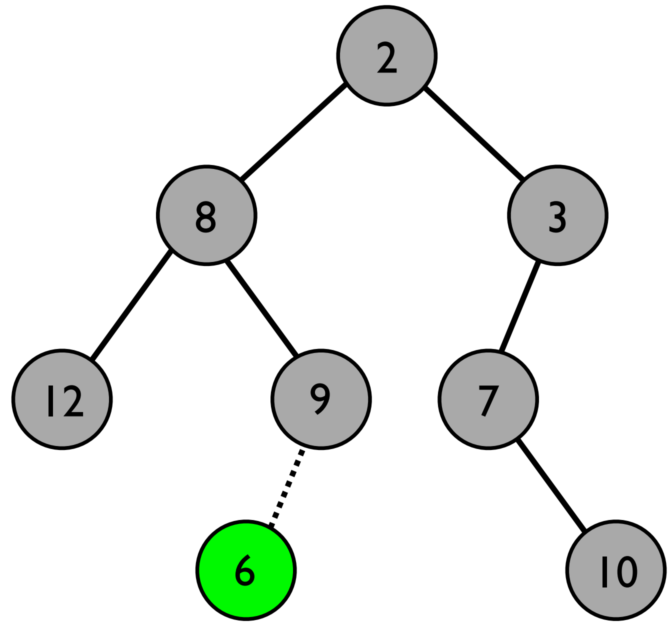
Heap – Find min



Heap – Insert

1. Add node as a leaf
(we'll see where later)

2. *"sift up:"* while current node is ~~>~~ < its parent, swap them.



Heap – Delete(*i*)

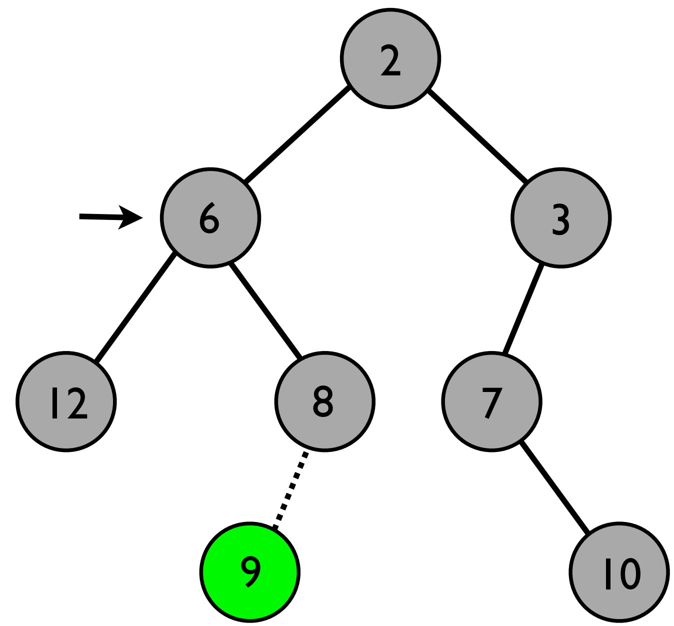
1. need a pointer to node containing key i

2. replace key to delete i with key j at a leaf node
(we'll see how to find a leaf soon)

3. Delete leaf

4. If $i \overset{>}{\cancel{<}} j$ then sift up, moving j up the tree.

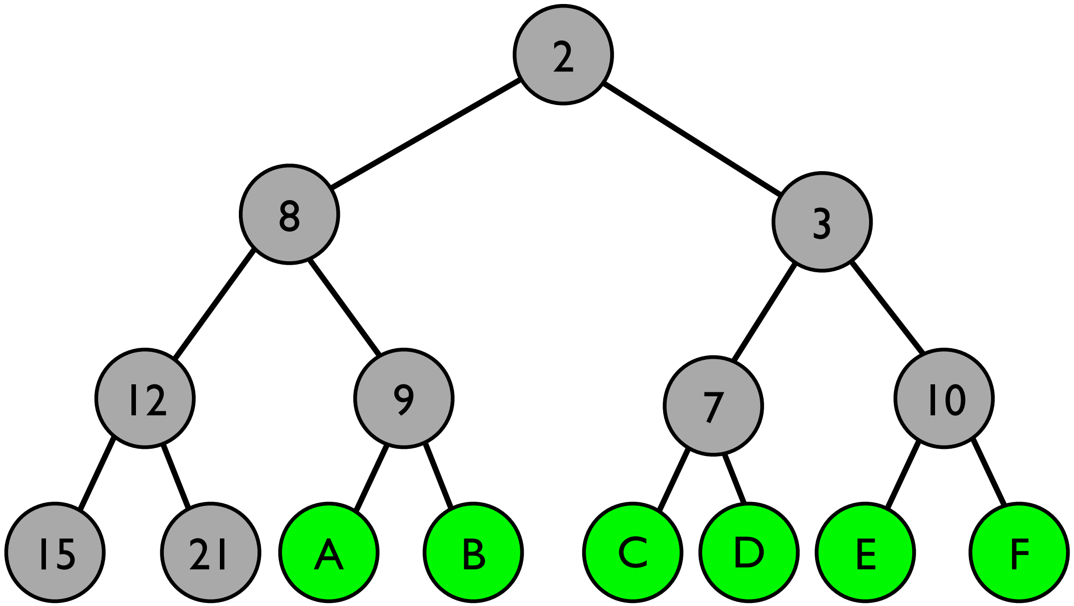
If $i \overset{<}{\cancel{>}} j$ then *"sift down"*: swap current node with **smallest of children** until its ~~bigger~~ **smaller** than all of its children.



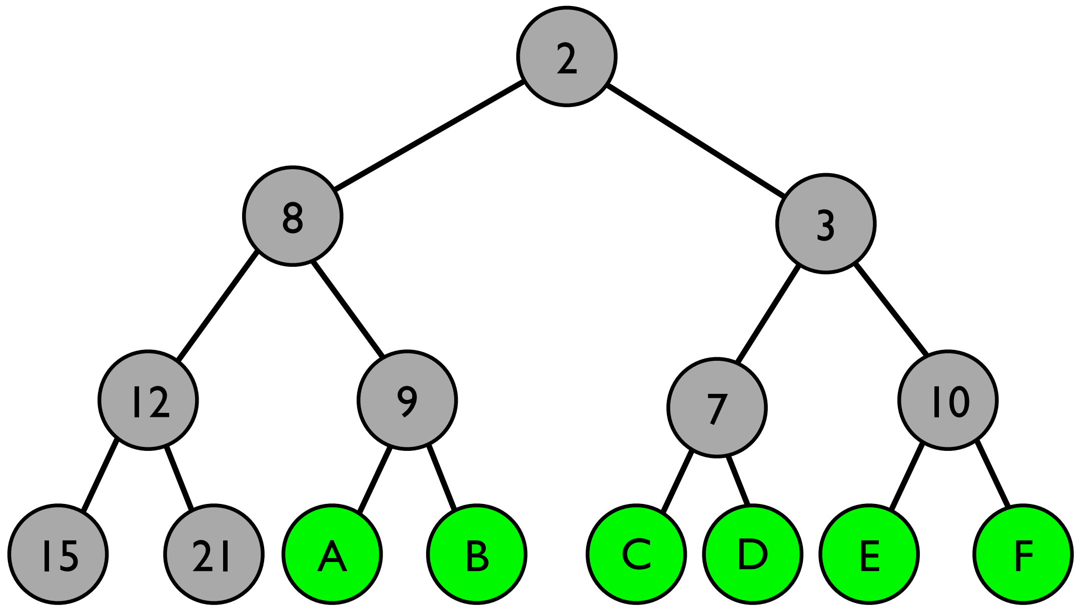
Time Complexity

- *findmin* takes O(1) time
- *insert*, *delete* take time O(tree height) plus the time to find the leaves.
- *deletemin*: same as delete
- But how do we find leaves used in *insert* and *delete*?
  - *delete*: use the last inserted node.
  - *insert*: choose node so tree remains complete.

Store Heap in a Complete Tree



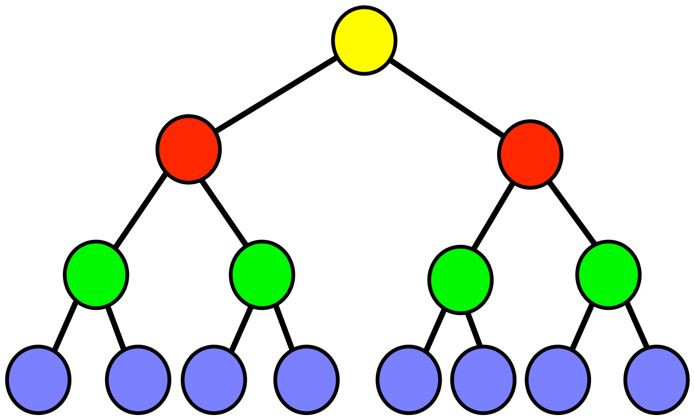
Store Heap in a Complete Tree



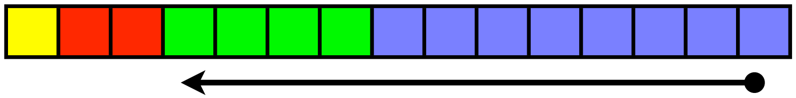
2	8	3	12	9	7	10	15	21	A	B	C	D	E	F
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

left(i): $2i$ **if** $2i \leq n$ **otherwise** 0
right(i): $(2i + 1)$ **if** $2i + 1 \leq n$ **otherwise** 0
parent(i): $\lfloor i/2 \rfloor$ **if** i ≥ 2 **otherwise** 0

Make Heap

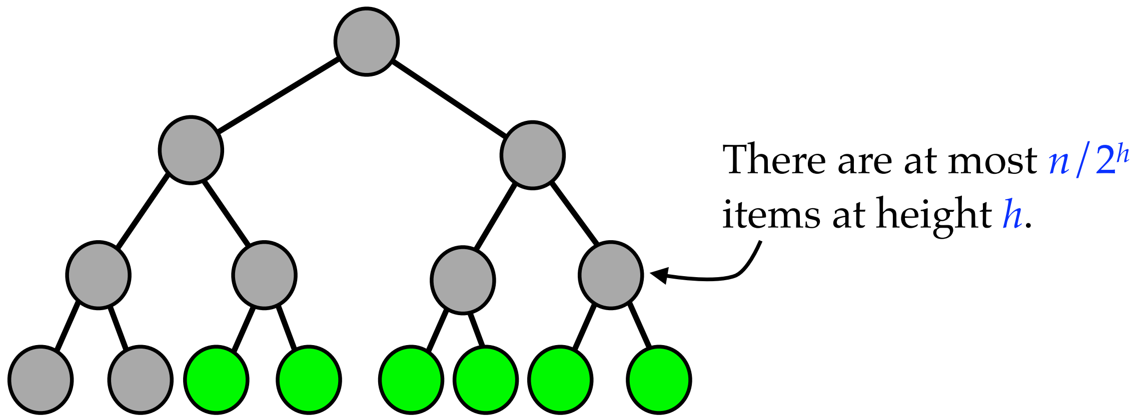


- n inserts gives a $O(n \log n)$ time bound.
- Better:
  - put items into array arbitrarily.
  - **for** $i = n \ldots 1$, *siftdown*(i).
- Each element trickles down to its correct place.



By the time you sift level i, all levels $i + 1$ and greater are already heap ordered.

Make Heap – Time Bound



Siftdown for all height h nodes is $O(h \bullet n/2^h)$ time

Total time

$= O(\sum_h h \bullet n/2^h)$ [sum of time for each height]

$= O(n \sum_h (h / 2^h))$ [factor out the n]

$= O(n)$ [sum bounded by const]

Heapsort – Another application of Heaps

Given unsorted array of integers

								end
8	2	12	10	7	15	21	9	3
1	2	3	4	5	6	7	8	9

makeheap – O(n)
Now first position has smallest item.

								end
2	8	3	12	9	7	10	15	21
1	2	3	4	5	6	7	8	9

Delete last item from heap.

							end	
21	8	3	12	9	7	10	15	2
1	2	3	4	5	6	7	8	9

siftdown new root key down

							end	
21	8	3	12	9	7	10	15	2
1	2	3	4	5	6	7	8	9

d-Heaps

- What about complete non-binary trees (e.g. every node has d children)?
  - *insert* takes $O(\log_d n)$ [because height $O(\log_d n)$]
  - *delete* takes $O(\boldsymbol{d} \log_d \mathrm{n})$ [why?]

- Can still store in an array.

- If you have few deletions, make d bigger so that tree is shorter.
- Can tune d to fit the relative proportions of inserts / deletes.

Outline: Heaps (I)

- Review of Binary Heaps
- Leftist Heaps
- Skew Heaps
- Amortized analysis
- Take-home messages

Leftist Heaps

Leftist Heaps

Target : **Speed up merging in O(*N*).**

Leftist Heaps

Target : Speed up merging in O(*N*).

- **Heap: Structure Property + Order Property**

Leftist Heaps

Target : Speed up merging in O(N).

- Heap: Structure Property + Order Property

Discussion 5: How fast can we merge two heaps if we simply use the original heap structure?

Leftist Heaps

Target : Speed up merging in O(N).

- Heap: Structure Property + Order Property

  - Have to copy one array into another ➡ $\Theta(N)$

Leftist Heaps

Target : Speed up merging in O(*N*).

- Heap: Structure Property + Order Property
  - Have to copy one array into another ➡ $\Theta(N)$
    - Use **pointers**

Leftist Heaps

Target : Speed up merging in O(N).

Heap: Structure Property + Order Property

Have to copy one array into another ➡ Θ(N)

Use pointers **Slow down all the operations**

Leftist Heaps

Target : **Speed up merging in O(N).**

Heap: Structure Property + Order Property

Have to copy one array into another ➡ $\Theta(N)$

Use pointers **Slow down all the operations**

Leftist Heap:

Order Property – the same

Structure Property – binary tree, but *unbalanced*

【Definition】The **null path length**, **Npl**(X), of any node X is the length of the shortest path from X to a node without two children. Define Npl(NULL) = –1.

【Definition】The null path length, Npl(X), of any node X is the length of the shortest path from X to a node without two children. Define Npl(NULL) = –1.

Note:

$$\text{Npl}(X) = \min \{ \text{Npl}(C) + 1 \text{ for all } C \text{ as children of } X \}$$

【Definition】The null path length, Npl(X), of any node X is the length of the shortest path from X to a node without two children. Define Npl(NULL) = –1.

Note:

$$\text{Npl}(X) = \min \{ \text{Npl}(C) + 1 \text{ for all } C \text{ as children of } X \}$$

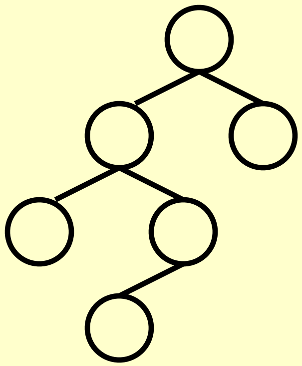
【Definition】The leftist heap property is that for every node X in the heap, the null path length of the left child is at least as large as that of the right child.

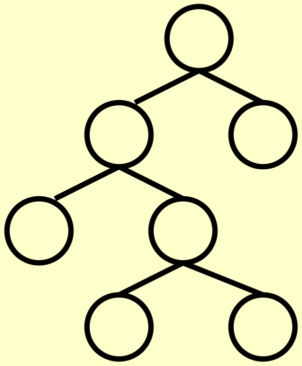
【Definition】The null path length, Npl(X), of any node X is the length of the shortest path from X to a node without two children. Define Npl(NULL) = –1.

Note:

$$\text{Npl}(X) = \min \{ \text{Npl}(C) + 1 \text{ for all } C \text{ as children of } X \}$$

【Definition】The leftist heap property is that for every node X in the heap, the null path length of the left child is at least as large as that of the right child.



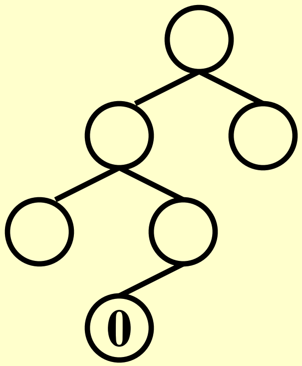


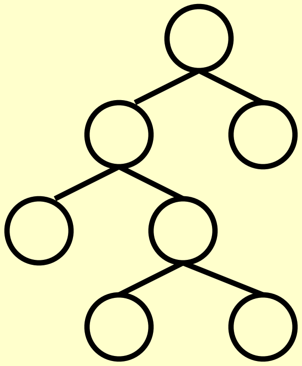
【Definition】The null path length, Npl(X), of any node X is the length of the shortest path from X to a node without two children. Define Npl(NULL) = –1.

Note:

$$\mathrm{Npl}(X) = \min \{ \mathrm{Npl}(C) + 1 \text{ for all } C \text{ as children of } X \}$$

【Definition】The leftist heap property is that for every node X in the heap, the null path length of the left child is at least as large as that of the right child.



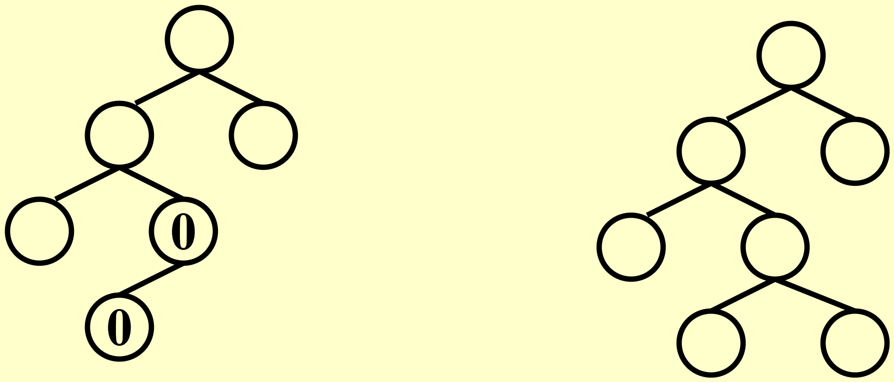


【Definition】The **null path length**, **Npl**(X), of any node X is the length of the shortest path from X to a node without two children. Define Npl(NULL) = –1.

Note:

$$\text{Npl}(X) = \min \{ \text{Npl}(C) + 1 \text{ for all } C \text{ as children of } X \}$$

【Definition】The **leftist heap property** is that for every node X in the heap, the null path length of the **left** child is **at least as large as** that of the **right** child.

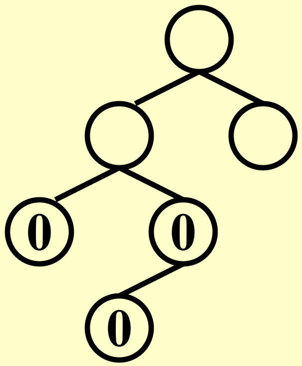


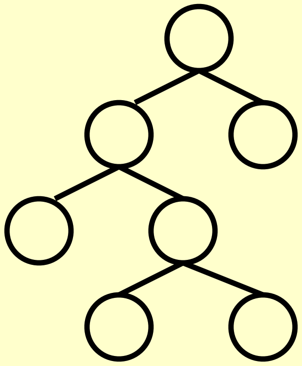
【Definition】The null path length, Npl(X), of any node X is the length of the shortest path from X to a node without two children. Define Npl(NULL) = –1.

Note:

$$Npl(X) = \min \{ Npl(C) + 1 \text{ for all } C \text{ as children of } X \}$$

【Definition】The leftist heap property is that for every node X in the heap, the null path length of the left child is at least as large as that of the right child.



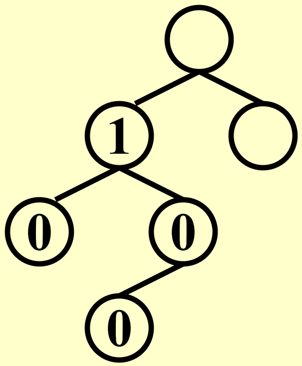


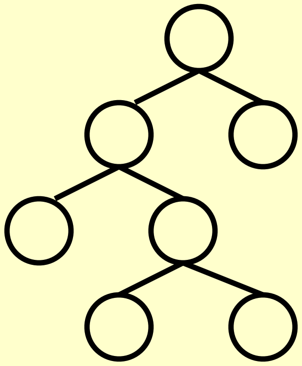
【Definition】The null path length, Npl(X), of any node X is the length of the shortest path from X to a node without two children. Define Npl(NULL) = –1.

Note:

$$Npl(X) = \min \{ Npl(C) + 1 \text{ for all } C \text{ as children of } X \}$$

【Definition】The leftist heap property is that for every node X in the heap, the null path length of the left child is at least as large as that of the right child.



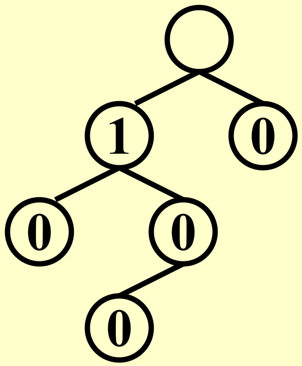


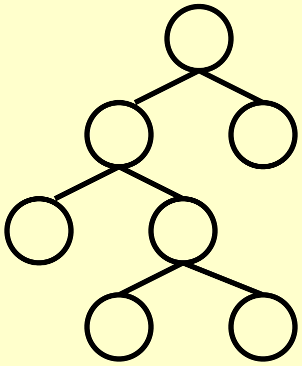
【Definition】The **null path length**, **Npl**(X), of any node X is the length of the shortest path from X to a node without two children. Define Npl(NULL) = –1.

Note:

$$\text{Npl}(X) = \min \{ \text{Npl}(C) + 1 \text{ for all } C \text{ as children of } X \}$$

【Definition】The **leftist heap property** is that for every node X in the heap, the null path length of the **left** child is **at least as large as** that of the **right** child.



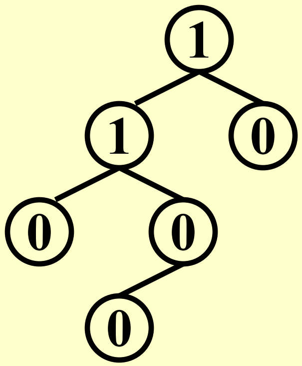


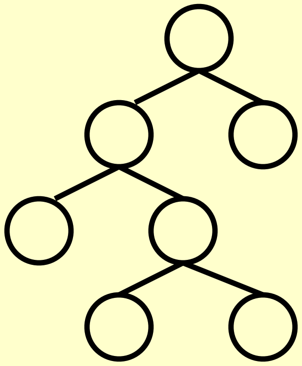
【Definition】The **null path length**, **Npl**(X), of any node X is the length of the shortest path from X to a node without two children. Define Npl(NULL) = –1.

Note:

$$\text{Npl}(X) = \min \{ \text{Npl}(C) + 1 \text{ for all } C \text{ as children of } X \}$$

【Definition】The **leftist heap property** is that for every node X in the heap, the null path length of the **left** child is **at least as large as** that of the **right** child.



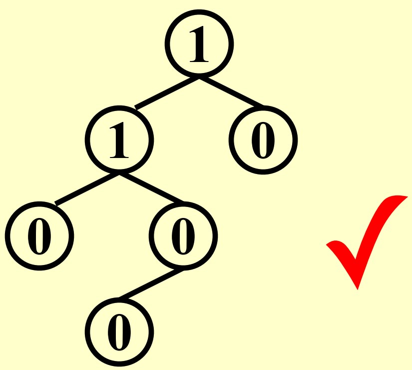


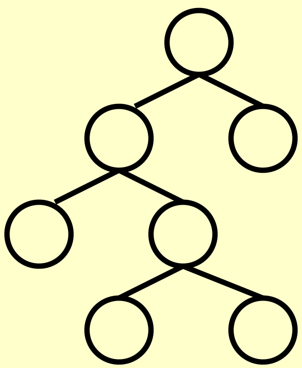
【Definition】The null path length, Npl(X), of any node X is the length of the shortest path from X to a node without two children. Define Npl(NULL) = –1.

Note:

$$\text{Npl}(X) = \min \{ \text{Npl}(C) + 1 \text{ for all } C \text{ as children of } X \}$$

【Definition】The leftist heap property is that for every node X in the heap, the null path length of the left child is at least as large as that of the right child.



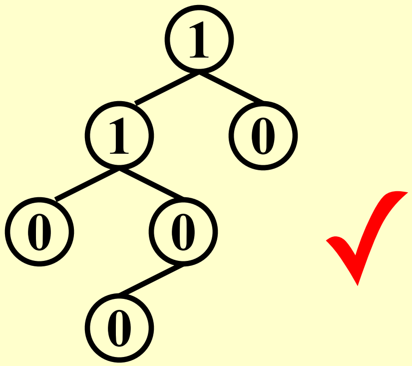


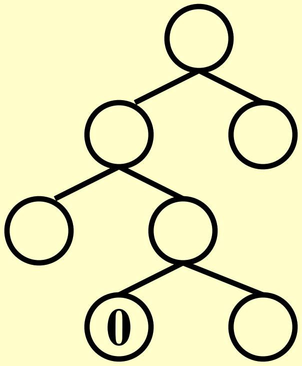
【Definition】The null path length, Npl(X), of any node X is the length of the shortest path from X to a node without two children. Define Npl(NULL) = –1.

Note:

$$\text{Npl}(X) = \min \{ \text{Npl}(C) + 1 \text{ for all } C \text{ as children of } X \}$$

【Definition】The leftist heap property is that for every node X in the heap, the null path length of the left child is at least as large as that of the right child.



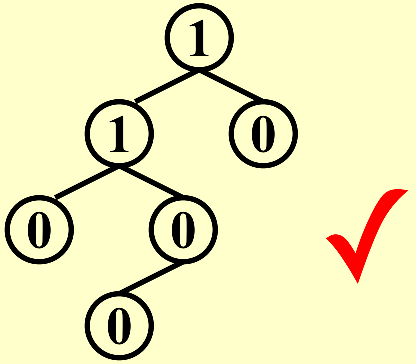


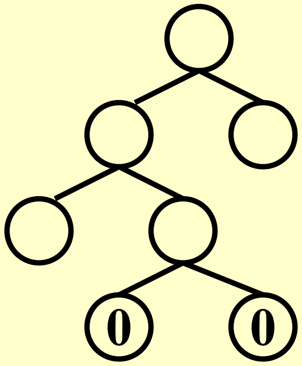
【Definition】The **null path length**, **Npl**(X), of any node X is the length of the shortest path from X to a node without two children. Define Npl(NULL) = –1.

Note:

$$\text{Npl}(X) = \min \{ \text{Npl}(C) + 1 \text{ for all } C \text{ as children of } X \}$$

【Definition】The **leftist heap property** is that for every node X in the heap, the null path length of the **left** child is **at least as large as** that of the **right** child.



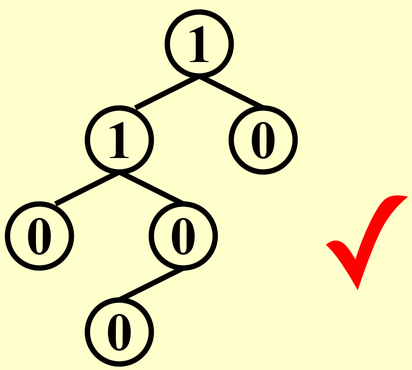


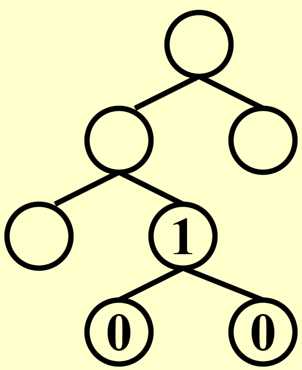
【Definition】The null path length, Npl(X), of any node X is the length of the shortest path from X to a node without two children. Define Npl(NULL) = –1.

Note:

$$\text{Npl}(X) = \min \{ \text{Npl}(C) + 1 \text{ for all } C \text{ as children of } X \}$$

【Definition】The leftist heap property is that for every node X in the heap, the null path length of the left child is at least as large as that of the right child.



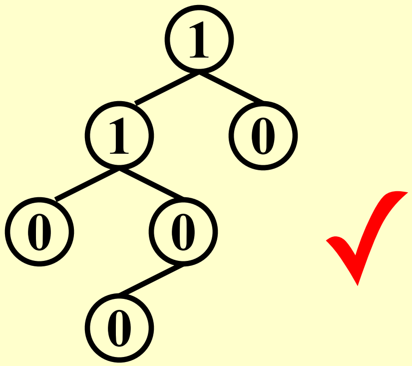


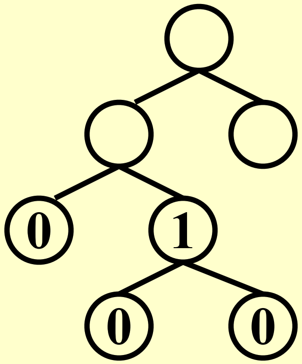
【Definition】The null path length, Npl(X), of any node X is the length of the shortest path from X to a node without two children. Define Npl(NULL) = –1.

Note:

$$\mathrm{Npl}(X) = \min \{ \mathrm{Npl}(C) + 1 \text{ for all } C \text{ as children of } X \}$$

【Definition】The leftist heap property is that for every node X in the heap, the null path length of the left child is at least as large as that of the right child.



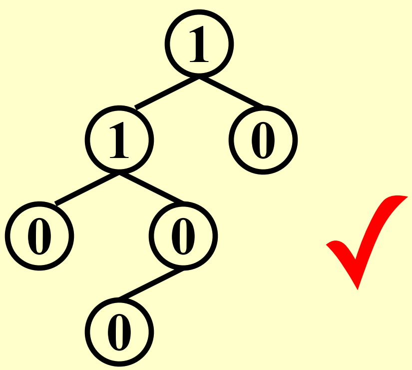


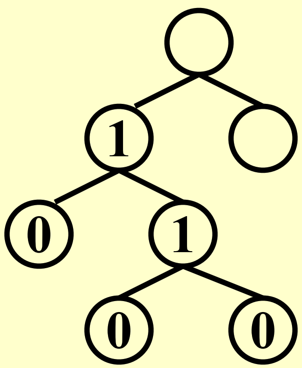
【Definition】The null path length, Npl(X), of any node X is the length of the shortest path from X to a node without two children. Define Npl(NULL) = –1.

Note:

$$\text{Npl}(X) = \min \{ \text{Npl}(C) + 1 \text{ for all } C \text{ as children of } X \}$$

【Definition】The leftist heap property is that for every node X in the heap, the null path length of the left child is at least as large as that of the right child.



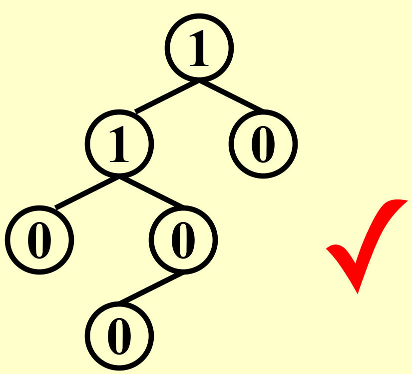


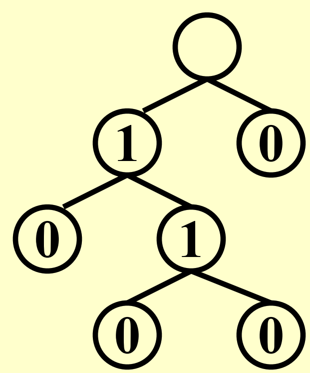
【Definition】The null path length, Npl(X), of any node X is the length of the shortest path from X to a node without two children. Define Npl(NULL) = –1.

Note:

$$\text{Npl}(X) = \min \{ \text{Npl}(C) + 1 \text{ for all } C \text{ as children of } X \}$$

【Definition】The leftist heap property is that for every node X in the heap, the null path length of the left child is at least as large as that of the right child.

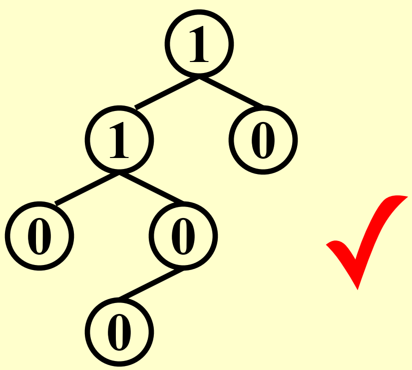


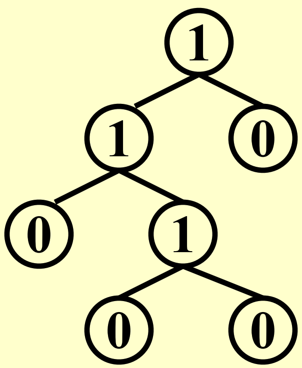


【Definition】The **null path length**, **Npl**(X), of any node X is the length of the shortest path from X to a node without two children. Define Npl(NULL) = –1.

> **Note:**
>
> Npl(X) = min { Npl(C) + 1 for all C as children of X }

【Definition】The **leftist heap property** is that for every node X in the heap, the null path length of the **left** child is **at least as large as** that of the **right** child.



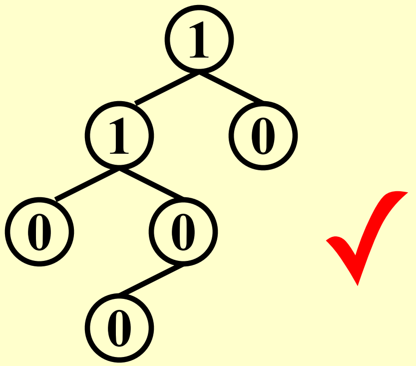


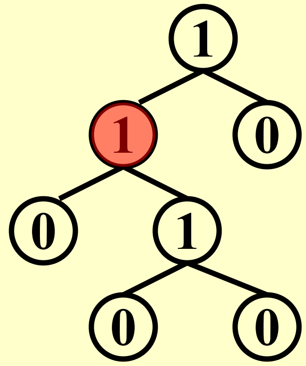
【Definition】The null path length, Npl(X), of any node X is the length of the shortest path from X to a node without two children. Define Npl(NULL) = –1.

Note:

$$\text{Npl}(X) = \min \{ \text{Npl}(C) + 1 \text{ for all } C \text{ as children of } X \}$$

【Definition】The leftist heap property is that for every node X in the heap, the null path length of the left child is at least as large as that of the right child.



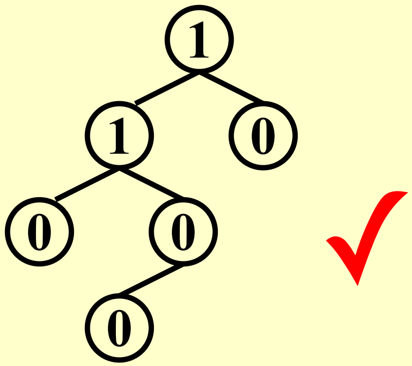


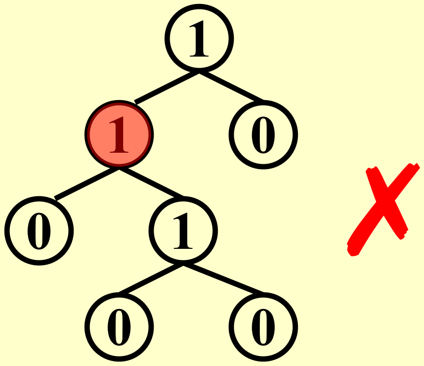
【Definition】The null path length, Npl(X), of any node X is the length of the shortest path from X to a node without two children. Define Npl(NULL) = –1.

Note:

$$\text{Npl}(X) = \min \{ \text{Npl}(C) + 1 \text{ for all } C \text{ as children of } X \}$$

【Definition】The leftist heap property is that for every node X in the heap, the null path length of the left child is at least as large as that of the right child.



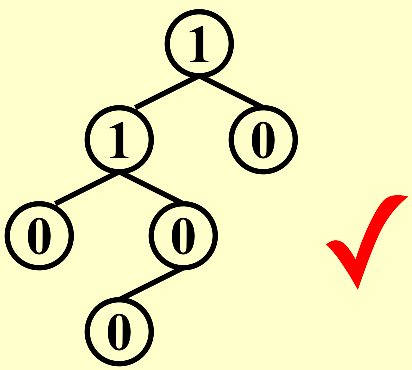


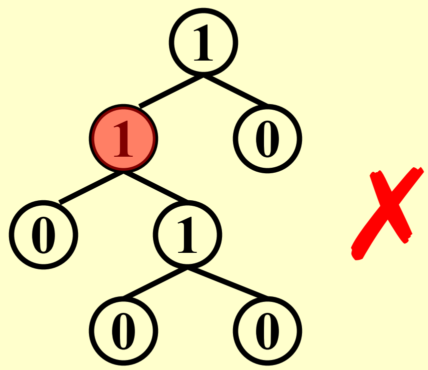
【Definition】The null path length, Npl(X), of any node X is the length of the shortest path from X to a node without two children. Define Npl(NULL) = –1.

Note:

$$\text{Npl}(X) = \min \{ \text{Npl}(C) + 1 \text{ for all } C \text{ as children of } X \}$$

【Definition】The leftist heap property is that for every node X in the heap, the null path length of the left child is at least as large as that of the right child.





The tree is biased to get deep toward the *left*.

【Theorem】A leftist tree with r nodes on the right path must have at least $2^r - 1$ nodes.

【Theorem】A leftist tree with r nodes on the right path must have at least $2^r - 1$ nodes.

Proof: By induction on p. 162.

【Theorem】A leftist tree with r nodes on the right path must have at least $2^r - 1$ nodes.

Proof: By induction on p. 162.

Discussion 6: How long is the right path of a leftist tree of *N* nodes? What does this conclusion mean to us?

【Theorem】A leftist tree with r nodes on the right path must have at least $2^r - 1$ nodes.

Proof: By induction on p. 162.

Note: The leftist tree of N nodes has a right path containing at most $\lfloor \log(N+1) \rfloor$ nodes.

【Theorem】A leftist tree with r nodes on the right path must have at least $2^r - 1$ nodes.

Proof: By induction on p. 162.

Note: The leftist tree of N nodes has a right path containing at most $\lfloor \log(N+1) \rfloor$ nodes.

We can perform all the work on the *right* path, which is guaranteed to be short.

【Theorem】A leftist tree with r nodes on the right path must have at least $2^r - 1$ nodes.

Proof: By induction on p. 162.

Note: The leftist tree of N nodes has a right path containing at most $\lfloor \log(N+1) \rfloor$ nodes.

We can perform all the work on the *right* path, which is guaranteed to be short.

Trouble makers: Insert and Merge

【Theorem】A leftist tree with r nodes on the right path must have at least $2^r - 1$ nodes.

Proof: By induction on p. 162.

Note: The leftist tree of N nodes has a right path containing at most $\lfloor \log(N+1) \rfloor$ nodes.

We can perform all the work on the *right* path, which is guaranteed to be short.

Trouble makers: Insert and Merge

Note: Insertion is merely a special case of merging.

Leftist trees have a short path

Thm. If rightmost path of leftist tree has r nodes, then whole tree has at least $2^r - 1$ nodes.

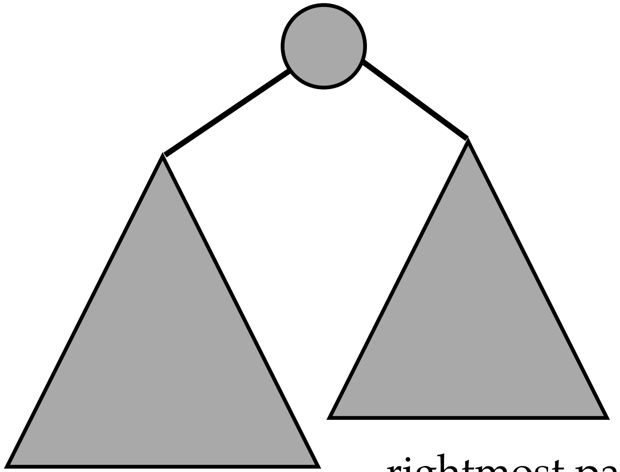
Proof.

Base Case: When $r = 1$, $2^1 - 1 = 1$ & tree has ≥ 1 node.

Induction hypothesis: Assume $N(i) \geq 2^i - 1$ for $i < r$.

Induction step: Left and right subtrees of the root have at least $2^{r-1} - 1$, nodes.

Thus, at least $2(2^{r-1} - 1) + 1 = 2^r - 1$ nodes in original tree. □



rightmost path length $\geq r - 1$

rightmost path length $= r - 1$

Therefore $n \geq 2^r - 1$, so r is O(log n)

☞ **Declaration:**

☞ **Declaration:**

```
struct TreeNode
{
    ElementType      Element;
    PriorityQueue    Left;
    PriorityQueue    Right;
    int              Npl;
} ;
```

☞ **Declaration:**

```
struct TreeNode
{
    ElementType     Element;
    PriorityQueue   Left;
    PriorityQueue   Right;
    int             Npl;
} ;
```

☞ **Declaration:**

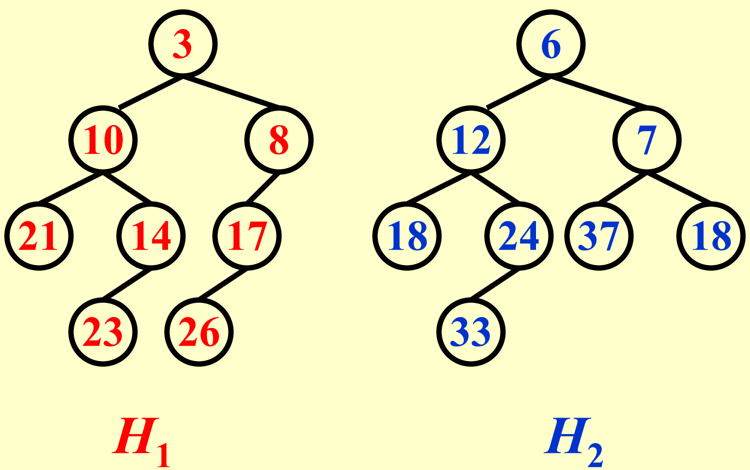
```
struct TreeNode
{
    ElementType     Element;
    PriorityQueue   Left;
    PriorityQueue   Right;
    int             Npl;
} ;
```

☞ **Merge (recursive version):**

☞ **Declaration:**

```
struct TreeNode
{
    ElementType      Element;
    PriorityQueue    Left;
    PriorityQueue    Right;
    int              Npl;
} ;
```

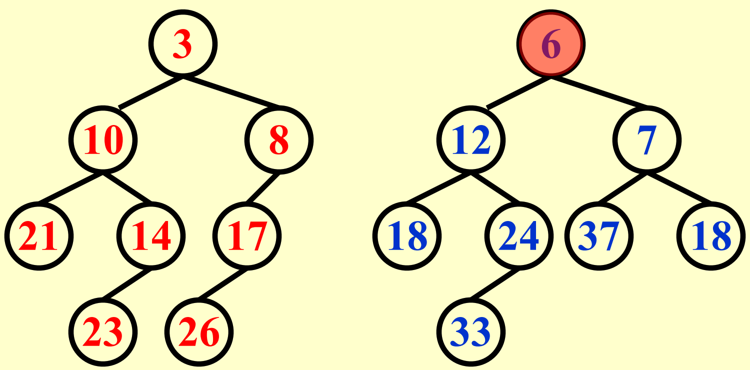
☞ **Merge (recursive version):**



☞ **Declaration:**

```
struct TreeNode
{
    ElementType     Element;
    PriorityQueue   Left;
    PriorityQueue   Right;
    int             Npl;
} ;
```

☞ **Merge (recursive version):**

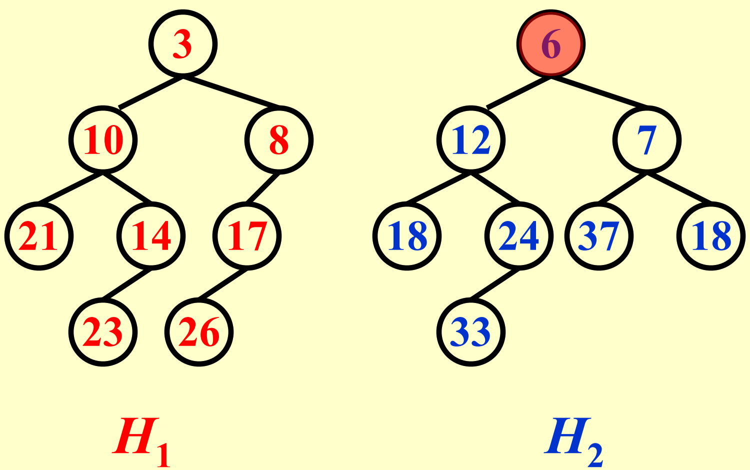


H_1 H_2

☞ **Declaration:**

```
struct TreeNode
{
    ElementType     Element;
    PriorityQueue   Left;
    PriorityQueue   Right;
    int             Npl;
} ;
```

☞ **Merge (recursive version):**



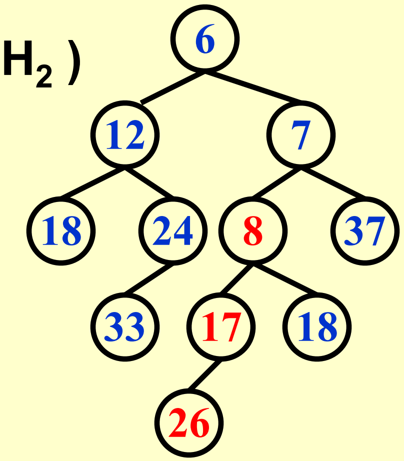
Step 1:

Merge(H_1->Right, H_2)

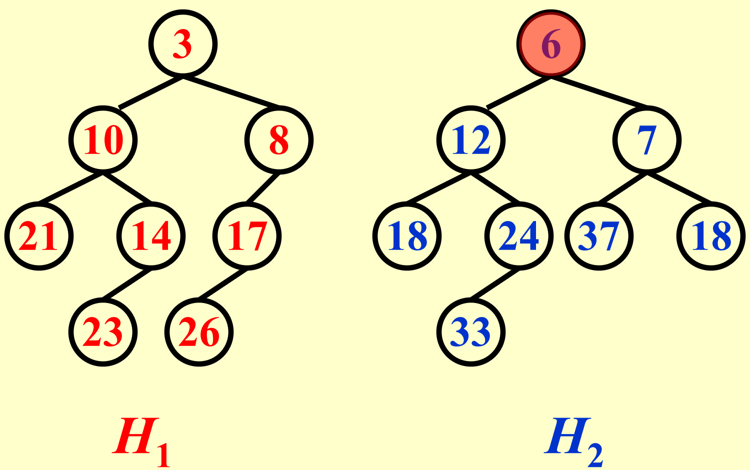
☞ **Declaration:**

```
struct TreeNode
{
    ElementType       Element;
    PriorityQueue     Left;
    PriorityQueue     Right;
    int               Npl;
} ;
```

***Step 1*:**
Merge(H_1->Right, H_2)



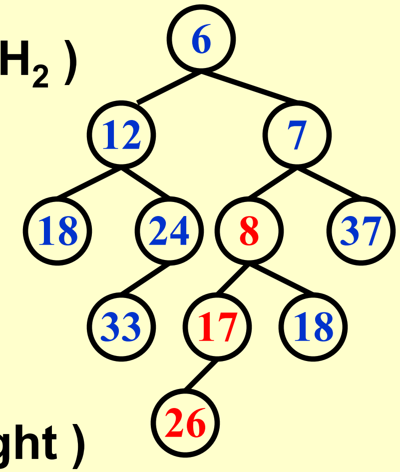
☞ **Merge (recursive version):**



☞ **Declaration:**

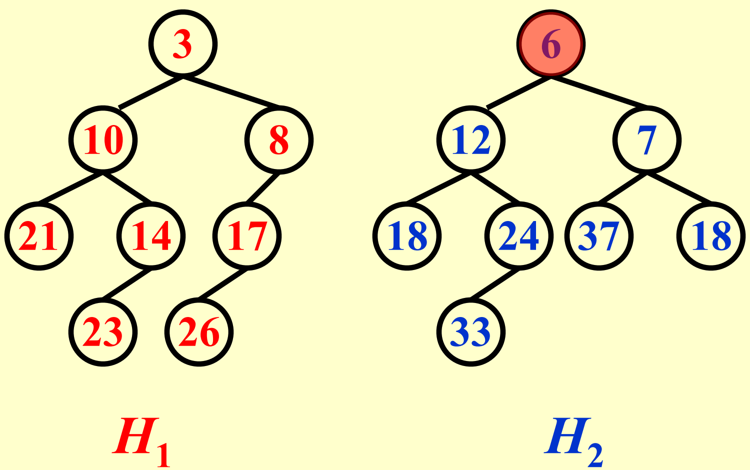
```
struct TreeNode
{
    ElementType     Element;
    PriorityQueue   Left;
    PriorityQueue   Right;
    int             Npl;
} ;
```

Step 1:
Merge(H_1->Right, H_2)



Step 2:
Attach(H_2, H_1->Right)

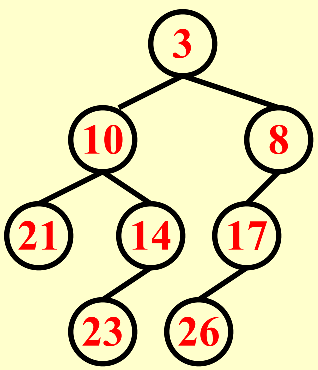
☞ **Merge (recursive version):**



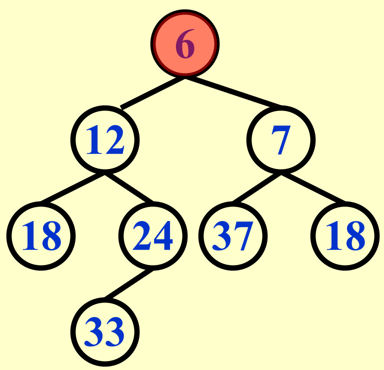
☞ **Declaration:**

```
struct TreeNode
{
    ElementType      Element;
    PriorityQueue    Left;
    PriorityQueue    Right;
    int              Npl;
} ;
```

☞ **Merge (recursive version):**

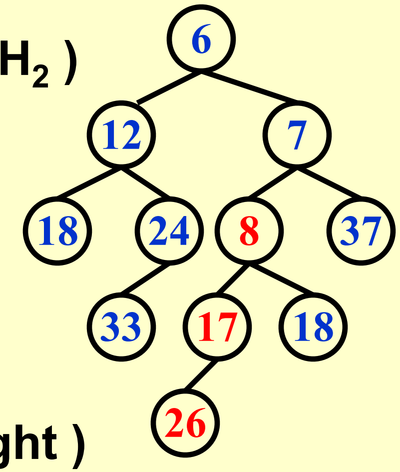


H_1

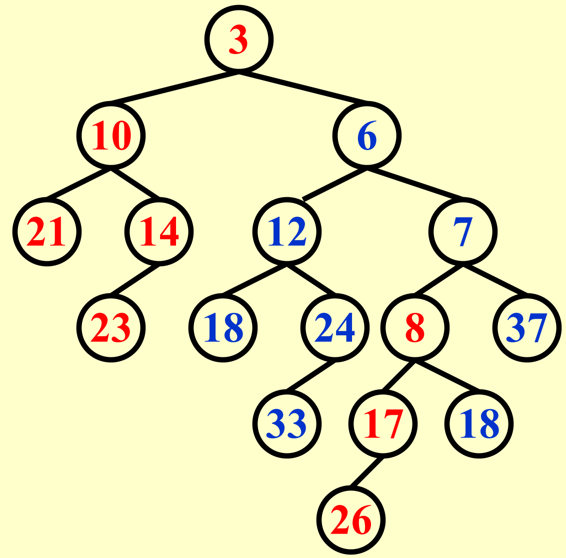


H_2

Step 1:
Merge(H_1->Right, H_2)



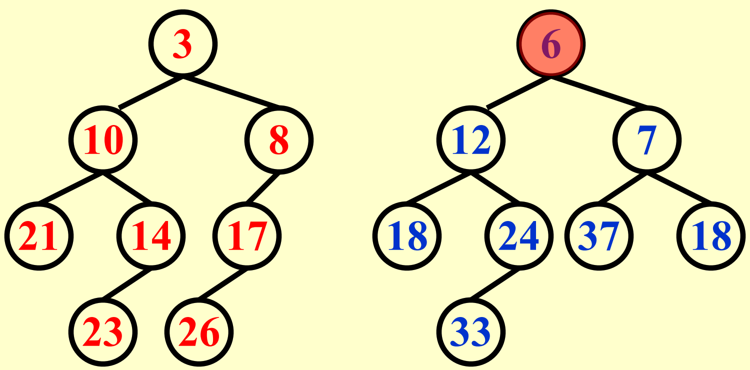
Step 2:
Attach(H_2, H_1->Right)



☞ Declaration:

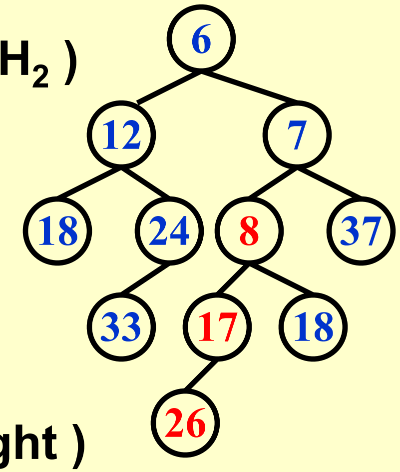
```
struct TreeNode
{
    ElementType     Element;
    PriorityQueue   Left;
    PriorityQueue   Right;
    int             Npl;
} ;
```

☞ Merge (recursive version):

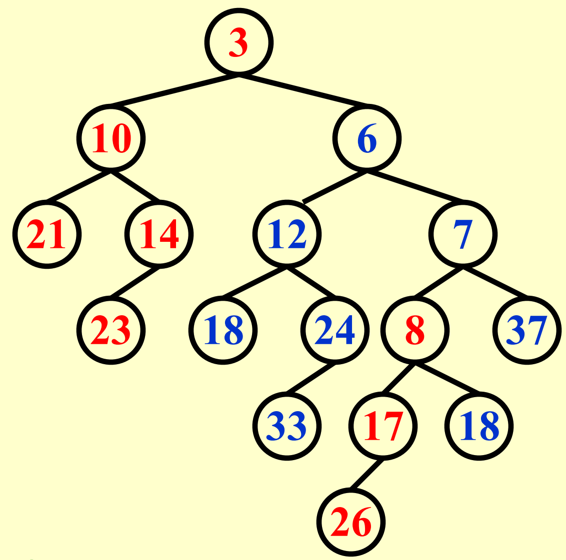


H_1　　　　*H_2*

Step 1:
Merge(H_1->Right, H_2)



Step 2:
Attach(H_2, H_1->Right)



Step 3:
Swap(H_1->Right, H_1->Left)
if necessary

```
PriorityQueue  Merge ( PriorityQueue H1, PriorityQueue H2 )
{
    if ( H1 == NULL )   return H2;
    if ( H2 == NULL )   return H1;
    if ( H1->Element < H2->Element )  return Merge1( H1, H2 );
    else return Merge1( H2, H1 );
}
```

```
PriorityQueue  Merge ( PriorityQueue H1, PriorityQueue H2 )
{
    if ( H1 == NULL )   return H2;
    if ( H2 == NULL )   return H1;
    if ( H1->Element < H2->Element )  return Merge1( H1, H2 );
    else return Merge1( H2, H1 );
}
```

```
static PriorityQueue
Merge1( PriorityQueue H1, PriorityQueue H2 )
{
    if ( H1->Left == NULL )     /* single node */
            H1->Left = H2;      /* H1->Right is already NULL
                                   and H1->Npl is already 0 */
    else {
            H1->Right = Merge( H1->Right, H2 );     /* Step 1 & 2 */
            if ( H1->Left->Npl < H1->Right->Npl )
                    SwapChildren( H1 );             /* Step 3 */
            H1->Npl = H1->Right->Npl + 1;
    } /* end else */
    return H1;
}
```

```
PriorityQueue  Merge ( PriorityQueue H1, PriorityQueue H2 )
{
    if ( H1 == NULL )   return H2;
    if ( H2 == NULL )   return H1;
    if ( H1->Element < H2->Element )  return Merge1( H1, H2 );
    else return Merge1( H2, H1 );
}
```

```
static PriorityQueue
Merge1( PriorityQueue H1, PriorityQueue H2 )
{
    if ( H1->Left == NULL )      /* single node */
            H1->Left = H2;       /* H1->Right is already NULL
                                    and H1->Npl is already 0 */
    else {
            H1->Right = Merge( H1->Right, H2 );     /* Step 1 & 2 */
            if ( H1->Left->Npl < H1->Right->Npl )
                    SwapChildren( H1 );             /* Step 3 */
            H1->Npl = H1->Right->Npl + 1;
    } /* end else */
    return H1;
}
```

What if *Npl* is NOT updated?

```
PriorityQueue  Merge ( PriorityQueue H1, PriorityQueue H2 )
{
    if ( H1 == NULL )   return H2;
    if ( H2 == NULL )   return H1;
    if ( H1->Element < H2->Element )  return Merge1( H1, H2 );
    else return Merge1( H2, H1 );
}
```

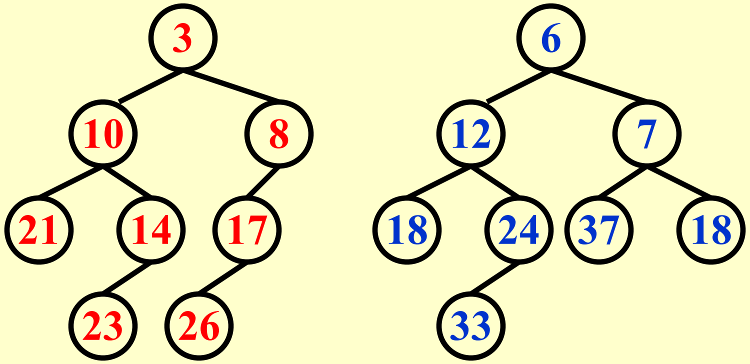
```
static PriorityQueue
Merge1( PriorityQueue H1, PriorityQueue H2 )
{
    if ( H1->Left == NULL )     /* single node */
            H1->Left = H2;      /* H1->Right is already NULL
                                   and H1->Npl is already 0 */
    else {
            H1->Right = Merge( H1->Right, H2 );     /* Step 1 & 2 */
            if ( H1->Left->Npl < H1->Right->Npl )
                    SwapChildren( H1 );            /* Step 3 */
            H1->Npl = H1->Right->Npl + 1;
    } /* end else */
    return H1;
}
```

$T_p = O(\log N)$

What if *Npl* is NOT updated?

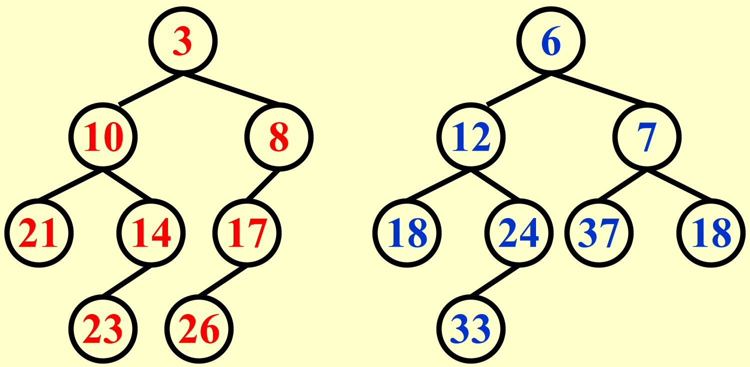
☞ **Merge (iterative version):**

☞ **Merge (iterative version):**



H_1 H_2

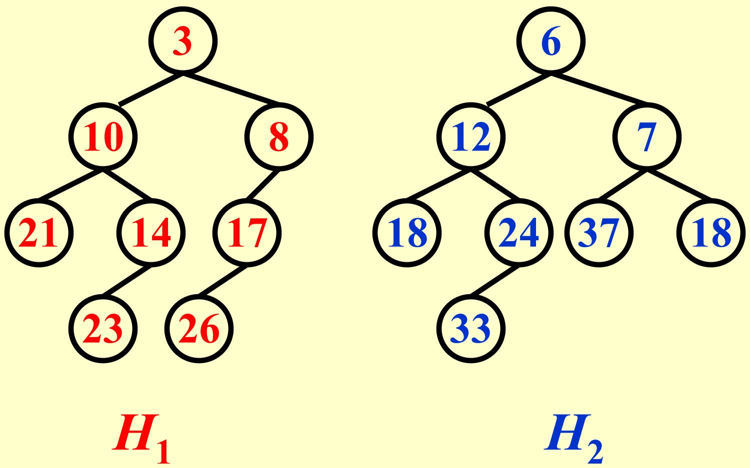
☞ **Merge (iterative version):**



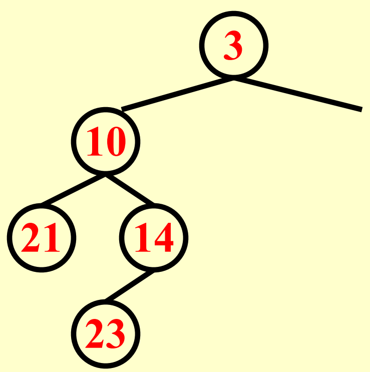
H_1 H_2

Step 1: **Sort the right paths without changing their left children**

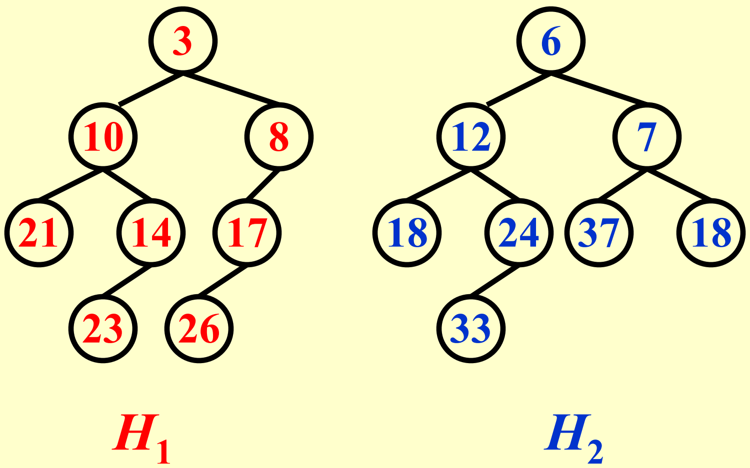
☞ **Merge (iterative version):**



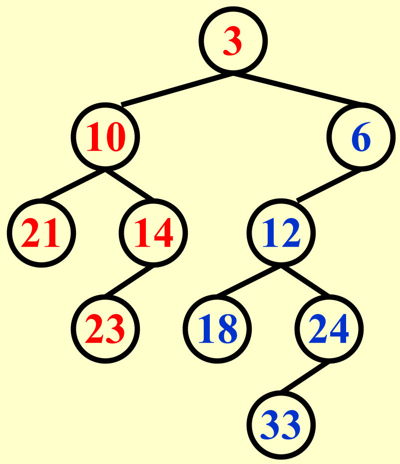
***Step 1*: Sort the right paths without changing their left children**



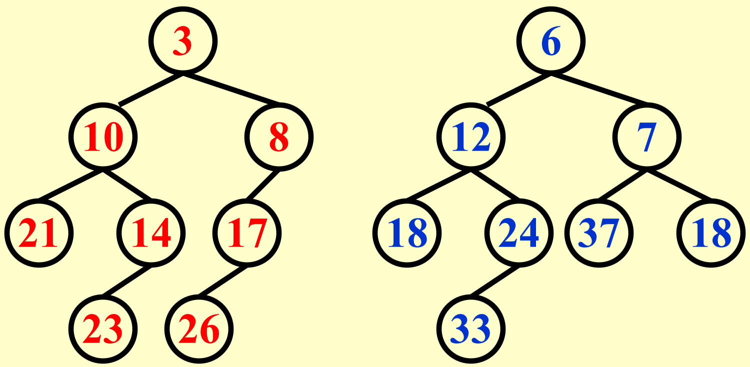
☞ **Merge (iterative version):**



Step 1: **Sort the right paths without changing their left children**

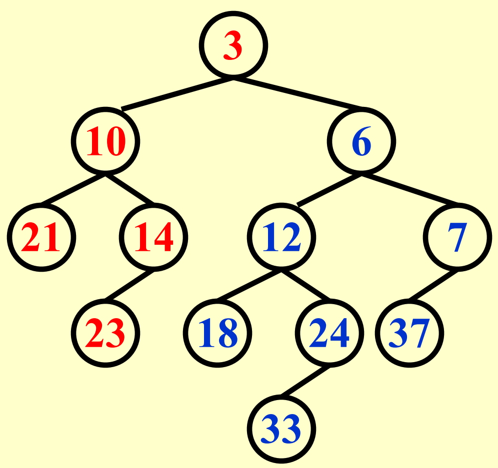


☞ **Merge (iterative version):**

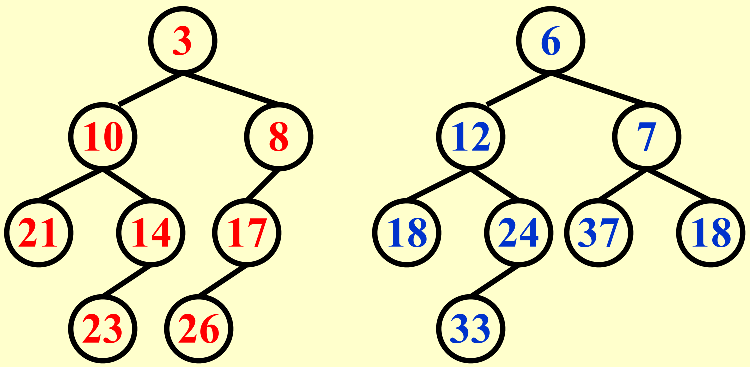


H_1 H_2

Step 1: **Sort the right paths without changing their left children**

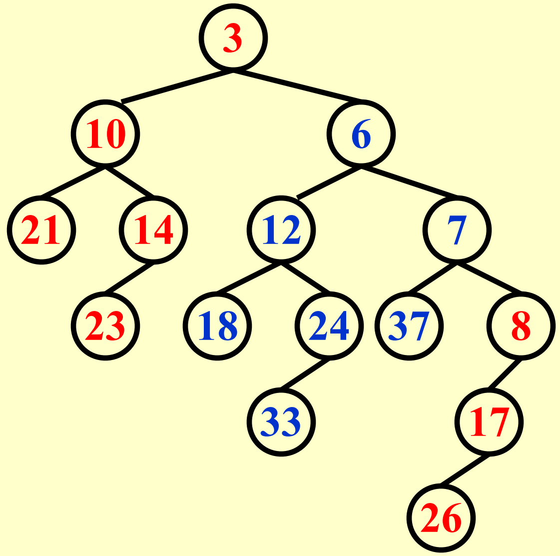


☞ **Merge (iterative version):**

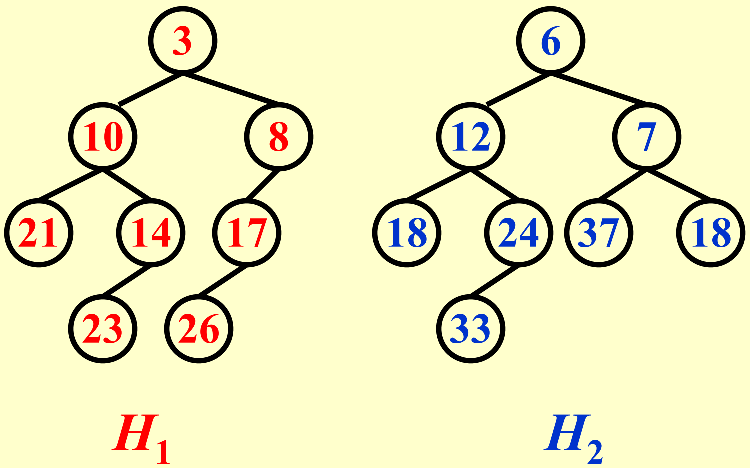


H_1 H_2

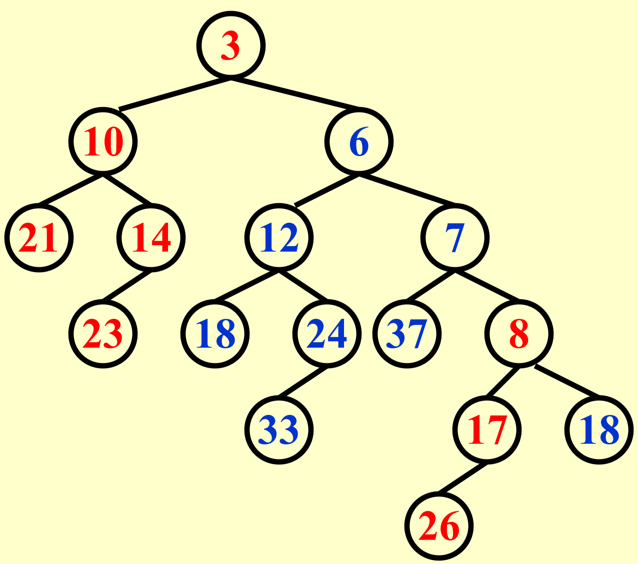
Step 1: **Sort the right paths without changing their left children**



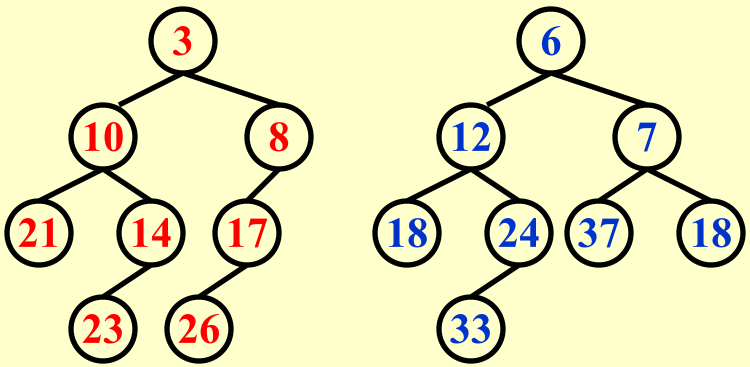
☞ **Merge (iterative version):**



Step 1: **Sort the right paths without changing their left children**

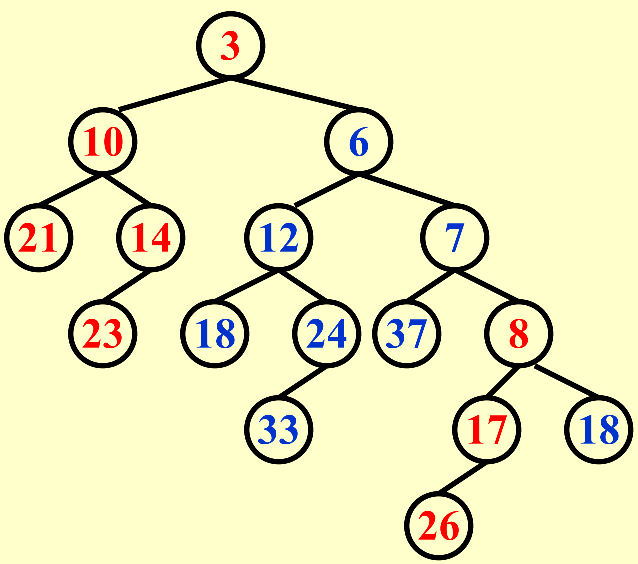


☞ **Merge (iterative version):**



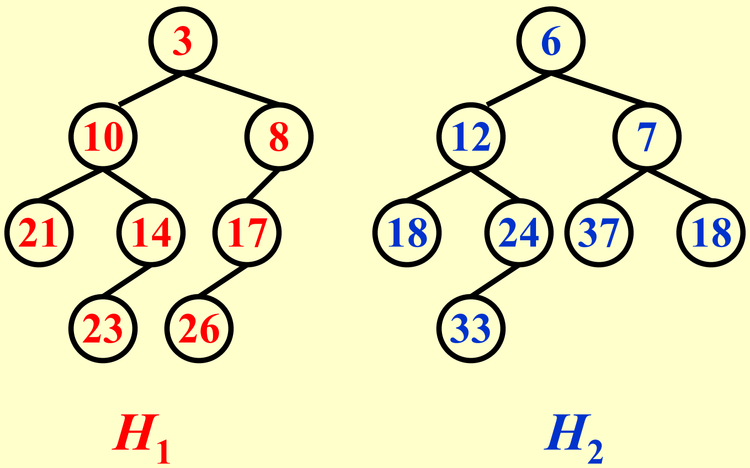
H_1 H_2

***Step 1*: Sort the right paths without changing their left children**



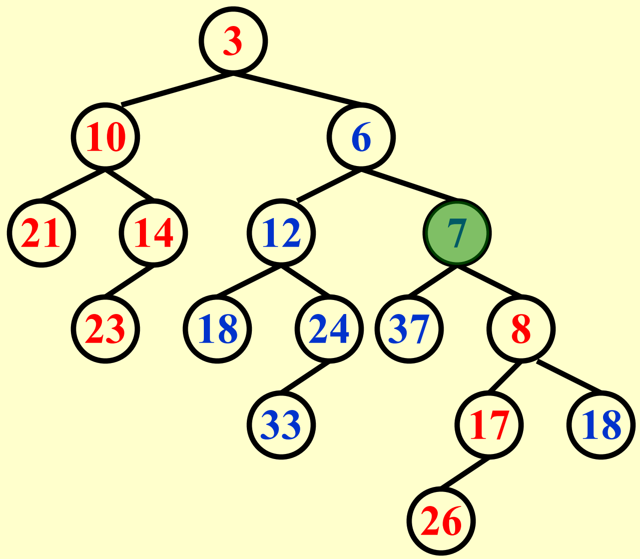
***Step 2*: Swap children if necessary**

☞ **Merge (iterative version):**

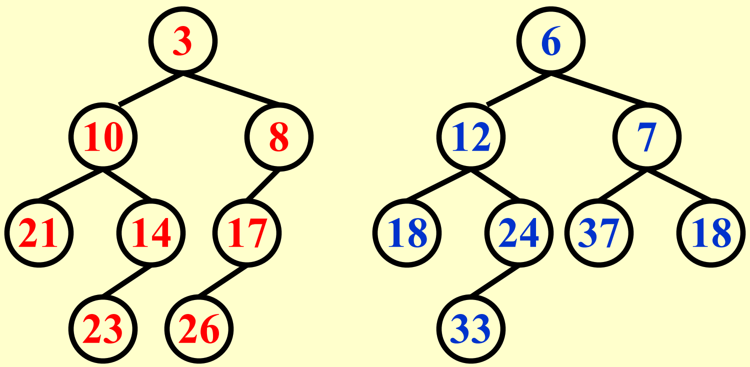


Step 2: **Swap children if necessary**

Step 1: **Sort the right paths without changing their left children**

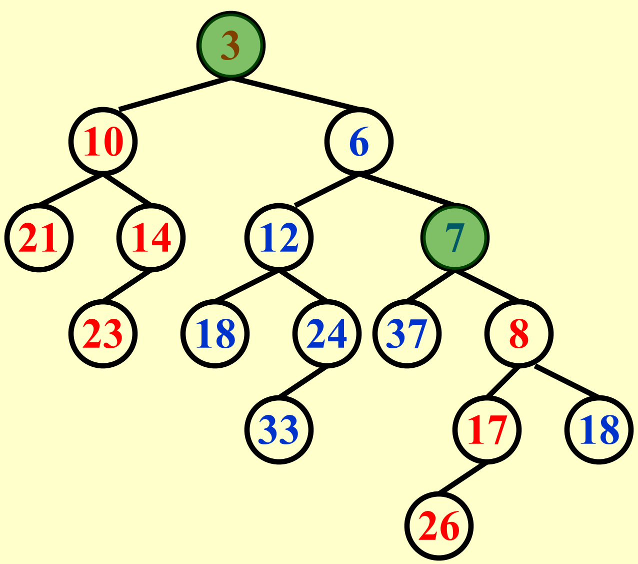


☞ **Merge (iterative version):**



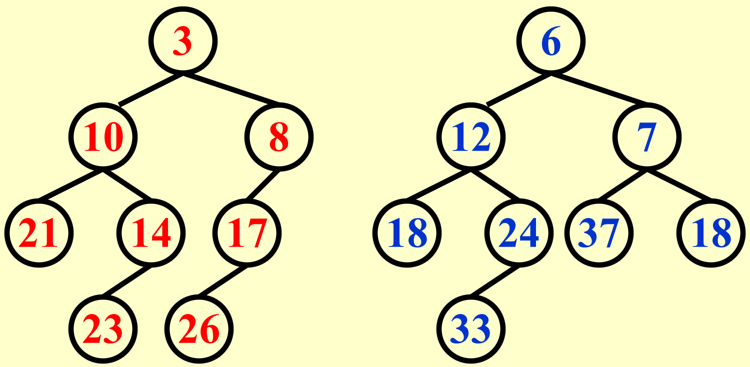
H_1 H_2

***Step 1*: Sort the right paths without changing their left children**



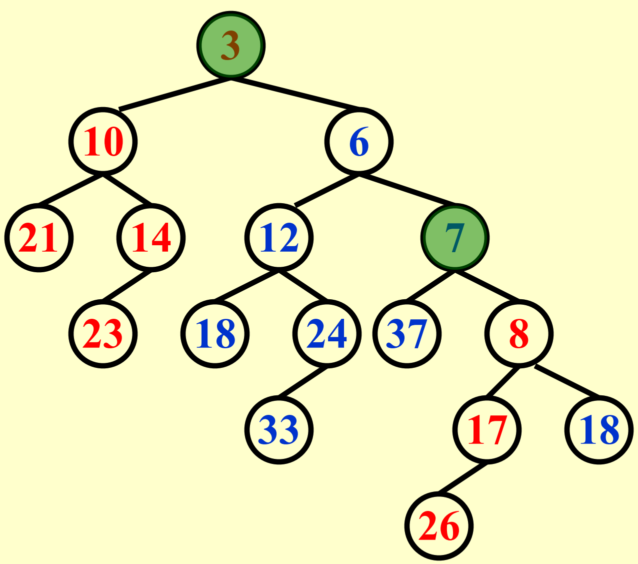
***Step 2*: Swap children if necessary**

☞ **Merge (iterative version):**

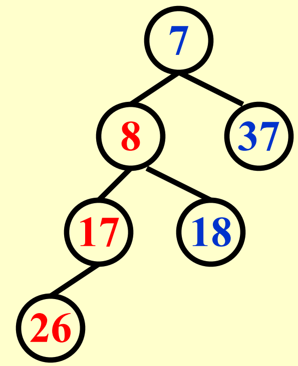


H_1 H_2

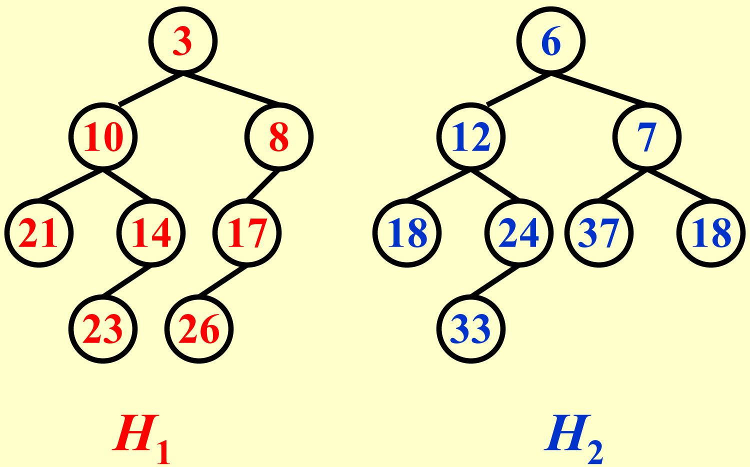
***Step 1*: Sort the right paths without changing their left children**



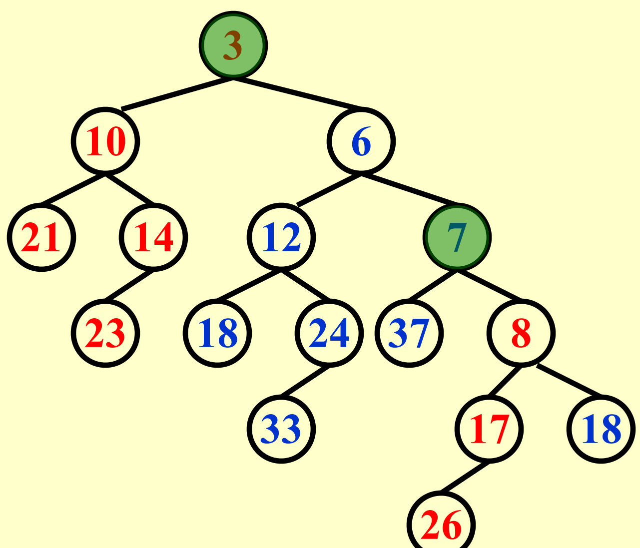
***Step 2*: Swap children if necessary**



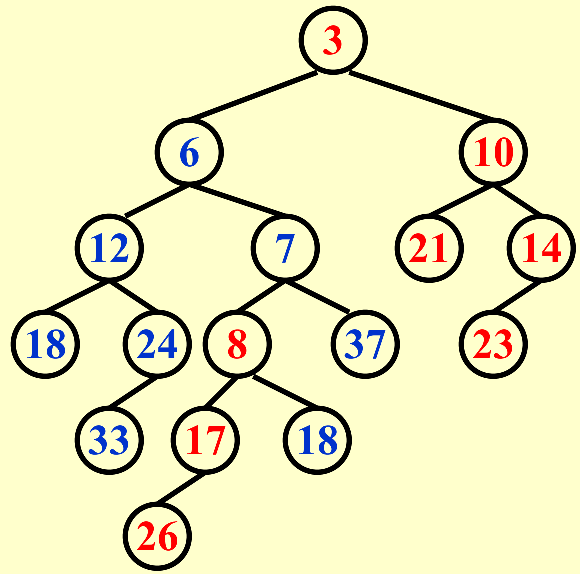
☞ **Merge (iterative version):**



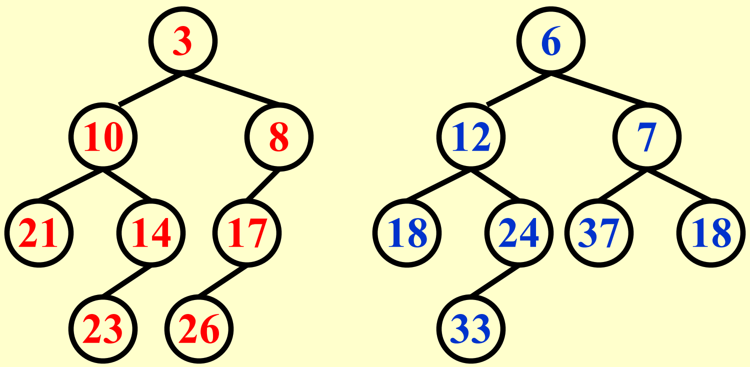
Step 1: **Sort the right paths without changing their left children**



Step 2: **Swap children if necessary**

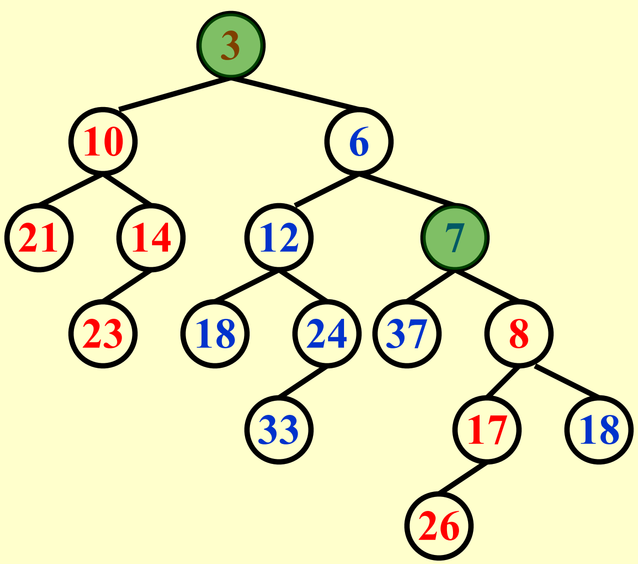


☞ **Merge (iterative version):**

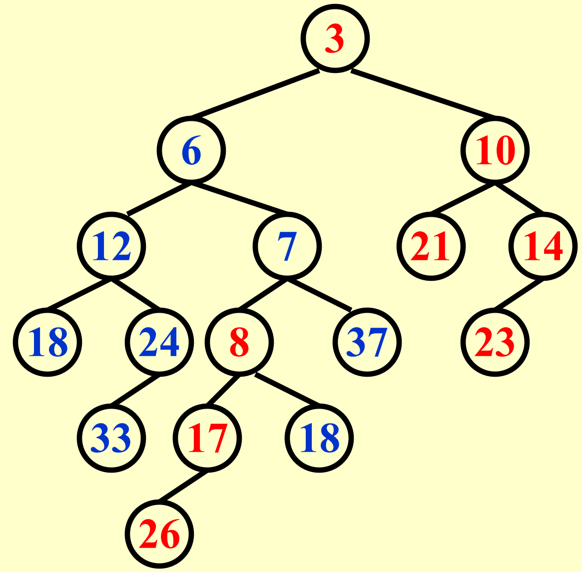


H_1 H_2

***Step 1*: Sort the right paths without changing their left children**

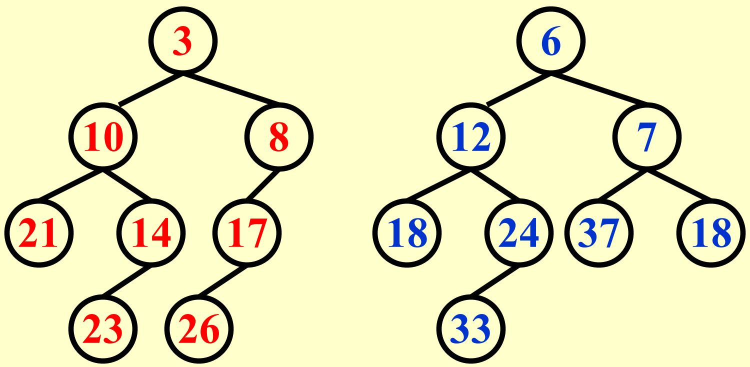


***Step 2*: Swap children if necessary**



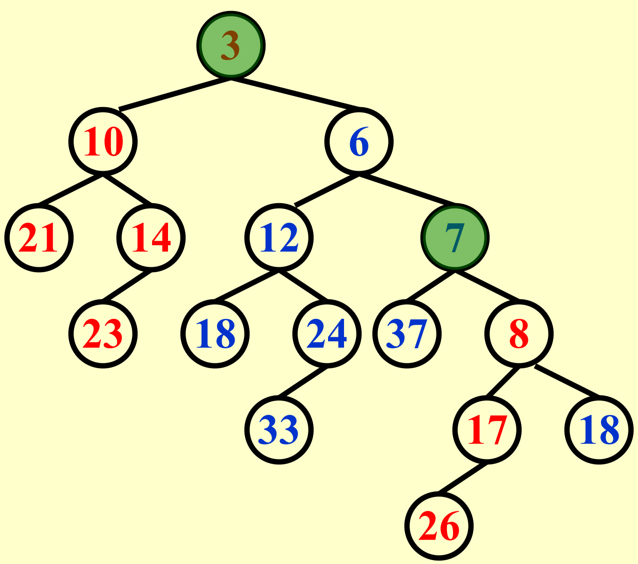
☞ **DeleteMin:**

☞ **Merge (iterative version):**

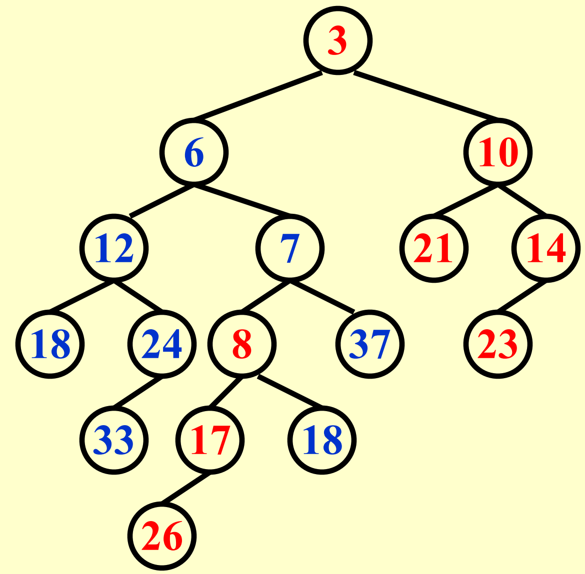


H_1 H_2

Step 1: **Sort the right paths without changing their left children**



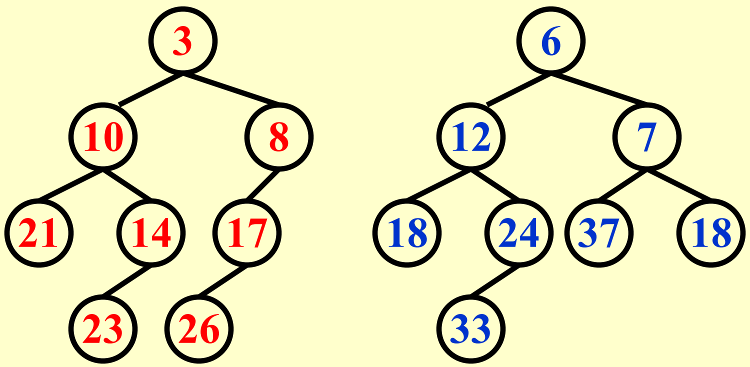
Step 2: **Swap children if necessary**



☞ **DeleteMin:**

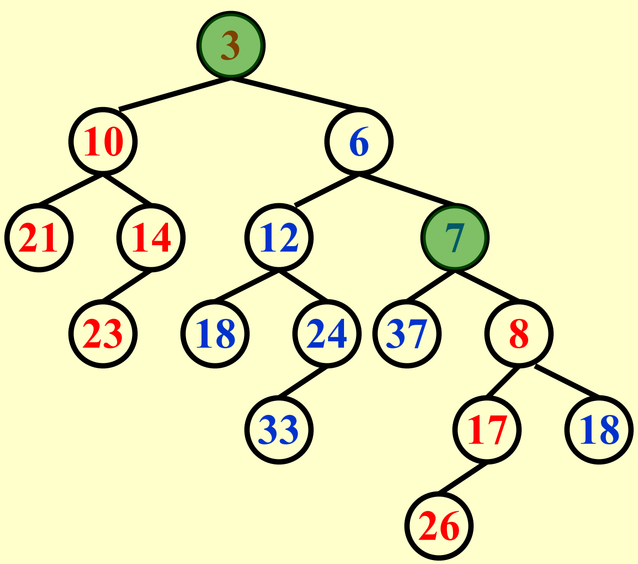
Step 1: **Delete the root**

☞ Merge (iterative version):

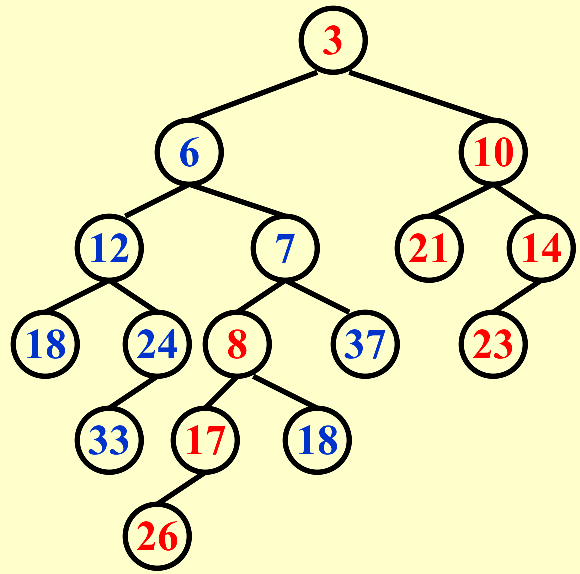


H_1 H_2

Step 1: **Sort the right paths without changing their left children**



Step 2: **Swap children if necessary**

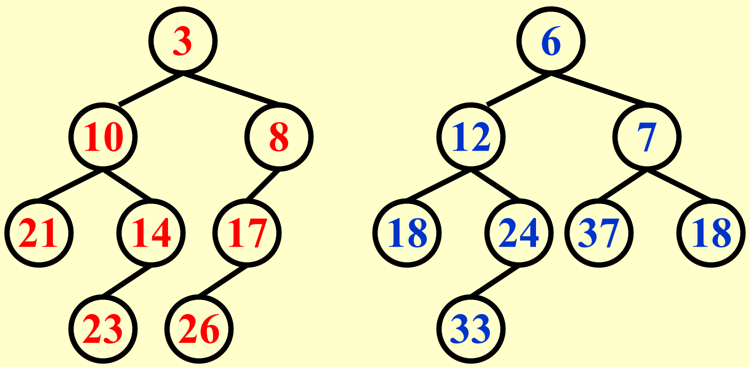


☞ DeleteMin:

Step 1: **Delete the root**

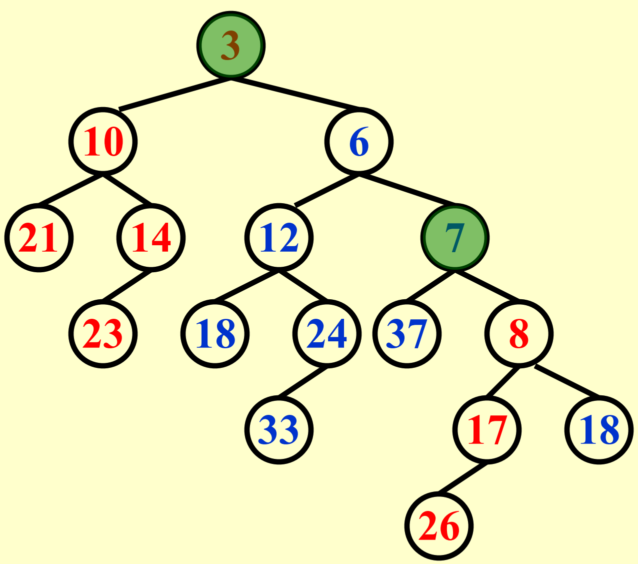
Step 2: **Merge the two subtrees**

☞ **Merge (iterative version):**

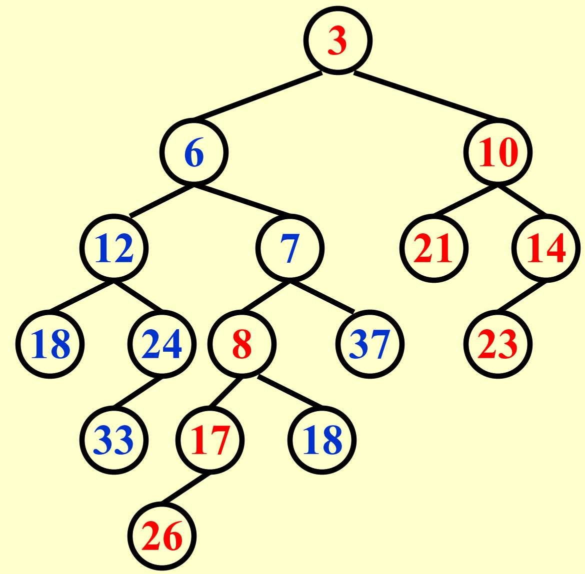


H_1 H_2

Step 1: **Sort the right paths without changing their left children**



Step 2: **Swap children if necessary**



☞ **DeleteMin:**

Step 1: **Delete the root**

Step 2: **Merge the two subtrees**

$T_p = O(\log N)$

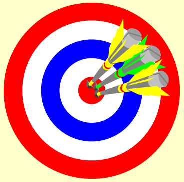
Outline: Heaps (I)

- Review of Binary Heaps
- Leftist Heaps
- Skew Heaps
- Amortized analysis
- Take-home messages

Skew Heaps

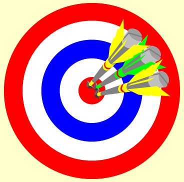
Skew Heaps -- a simple version of the leftist heaps

Skew Heaps -- a simple version of the leftist heaps



Target : **Any M consecutive operations take at most O(M log N) time.**

Skew Heaps -- a simple version of the leftist heaps



Target : **Any M consecutive operations take at most $O(M \log N)$ time.**

☞ **Merge: Always swap the left and right children except that the largest of all the nodes on the right paths does not have its children swapped. No Npl.**

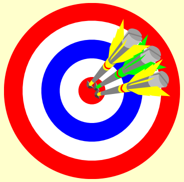
Skew Heaps -- a simple version of the leftist heaps

Target : Any M consecutive operations take at most $O(M \log N)$ time.

☞ **Merge:** **Always** swap the left and right children except that the **largest** of all the nodes on the right paths does not have its children swapped. **No Npl.**

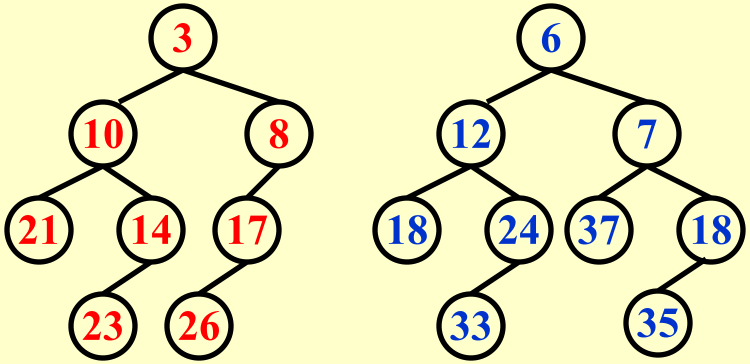
Not really a special case, but a natural stop in the recursions.

Skew Heaps -- a simple version of the leftist heaps



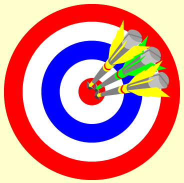
Target : Any M consecutive operations take at most $O(M \log N)$ time.

☞ **Merge: Always swap the left and right children except that the largest of all the nodes on the right paths does not have its children swapped. No Npl.**



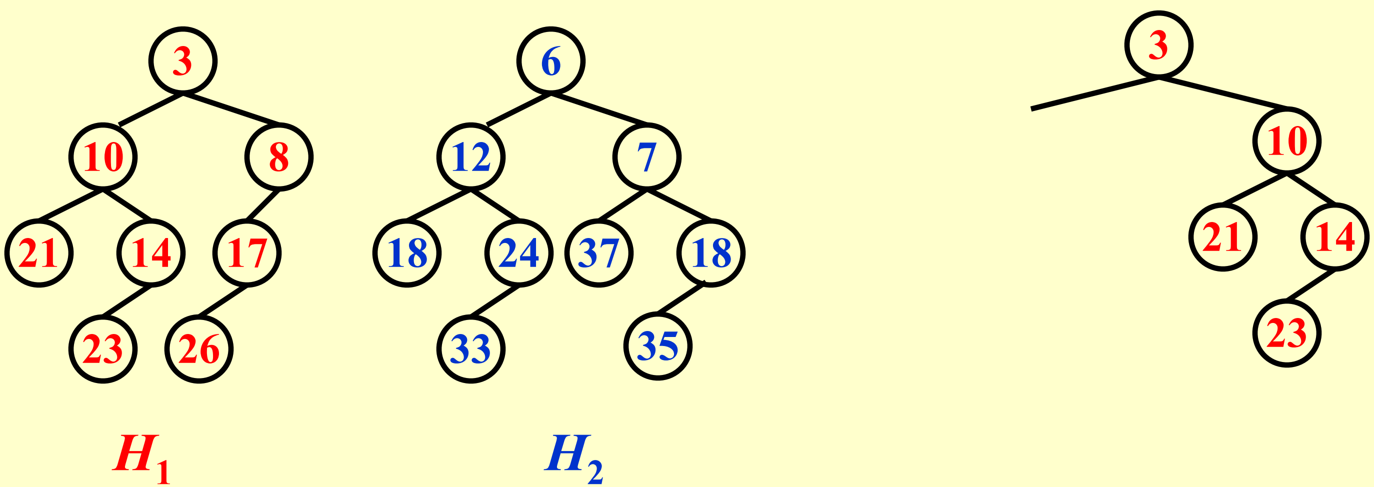
H_1 H_2

Skew Heaps -- a simple version of the leftist heaps



Target : Any M consecutive operations take at most $O(M \log N)$ time.

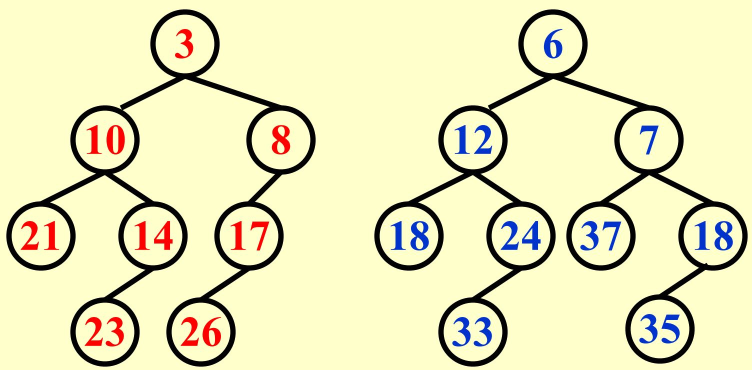
☞ **Merge:** **Always** swap the left and right children except that the **largest** of all the nodes on the right paths does not have its children swapped. **No Npl.**



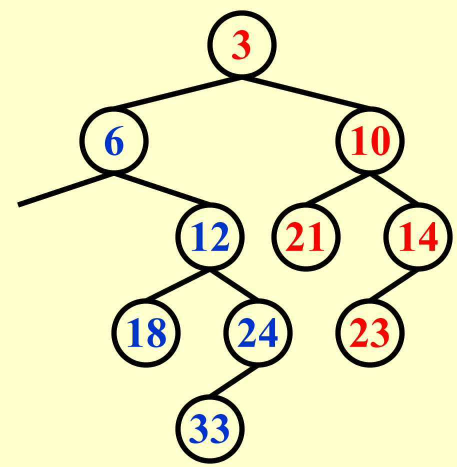
Skew Heaps -- a simple version of the leftist heaps

Target : Any M consecutive operations take at most $O(M \log N)$ time.

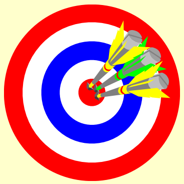
☞ **Merge: Always swap the left and right children except that the largest of all the nodes on the right paths does not have its children swapped. No Npl.**



H_1 H_2

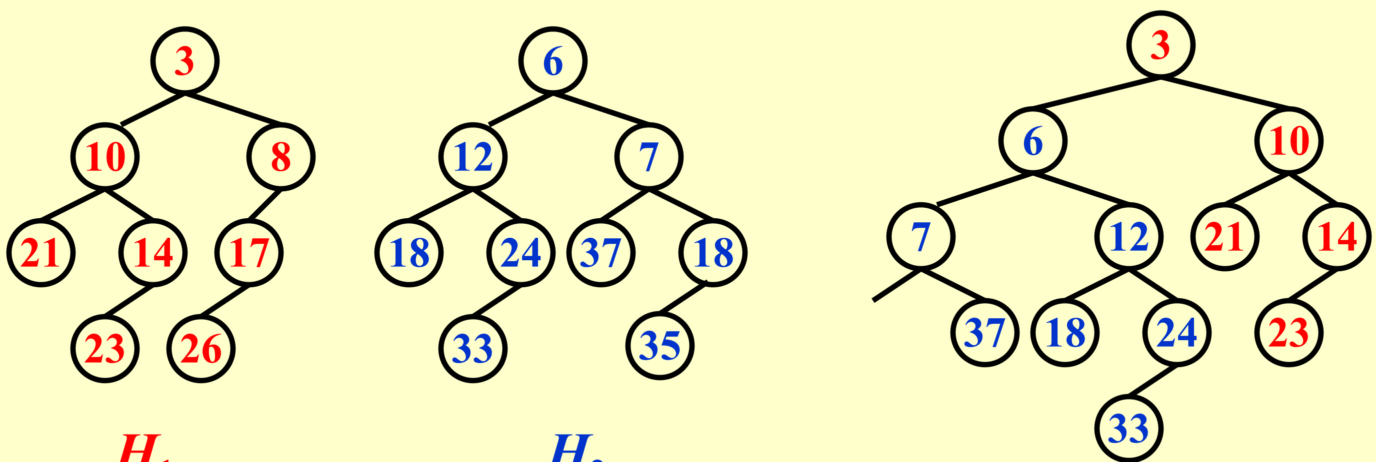


Skew Heaps -- a simple version of the leftist heaps



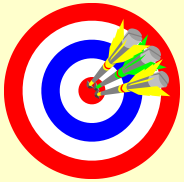
Target : Any M consecutive operations take at most $O(M \log N)$ time.

☞ **Merge:** **Always** swap the left and right children except that the **largest** of all the nodes on the right paths does not have its children swapped. **No Npl.**



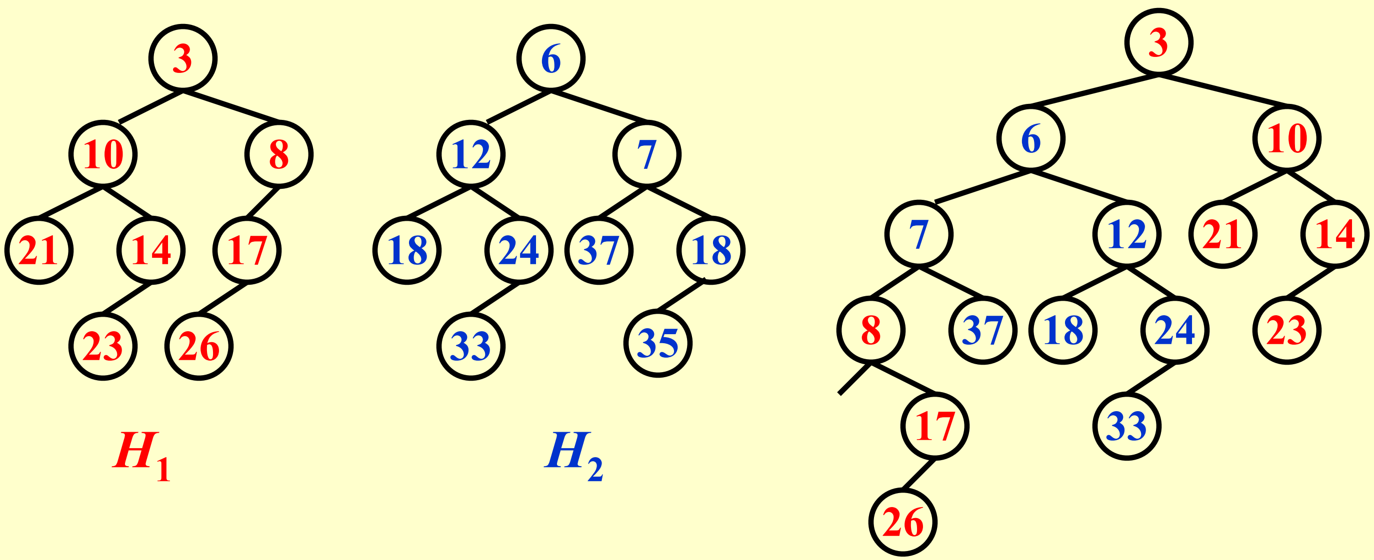
H_1 H_2

Skew Heaps -- a simple version of the leftist heaps

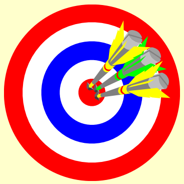


Target : Any M consecutive operations take at most $O(M \log N)$ time.

☞ **Merge: Always swap the left and right children except that the largest of all the nodes on the right paths does not have its children swapped. No Npl.**

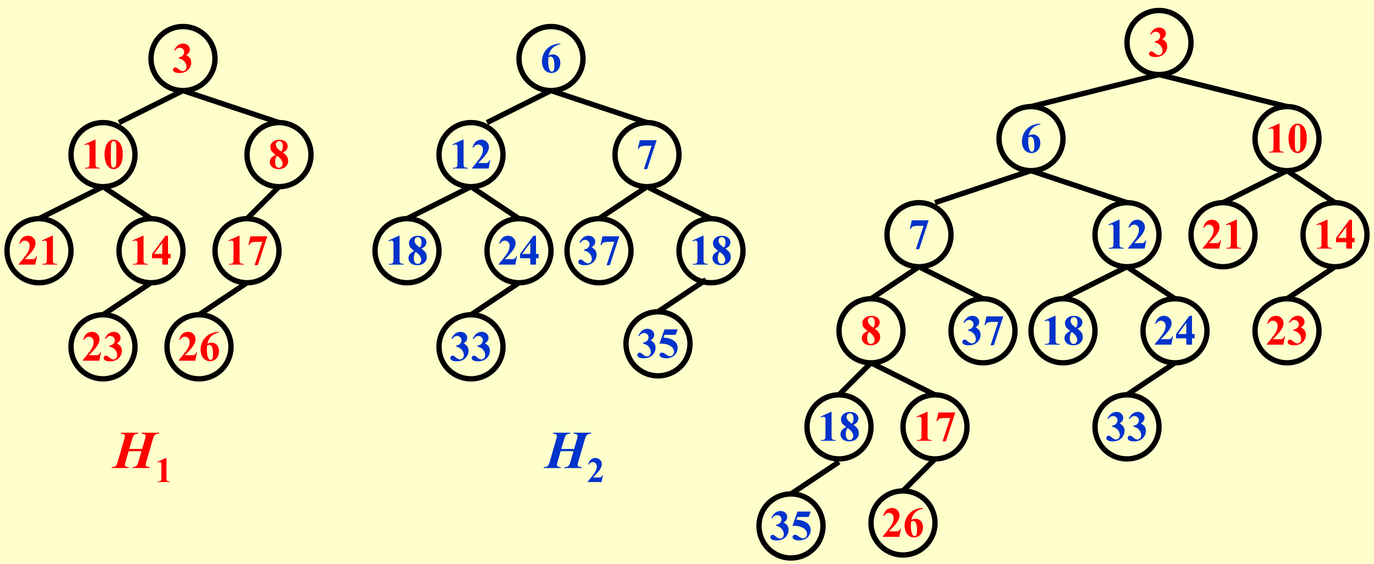


Skew Heaps -- a simple version of the leftist heaps



Target : Any M consecutive operations take at most $O(M \log N)$ time.

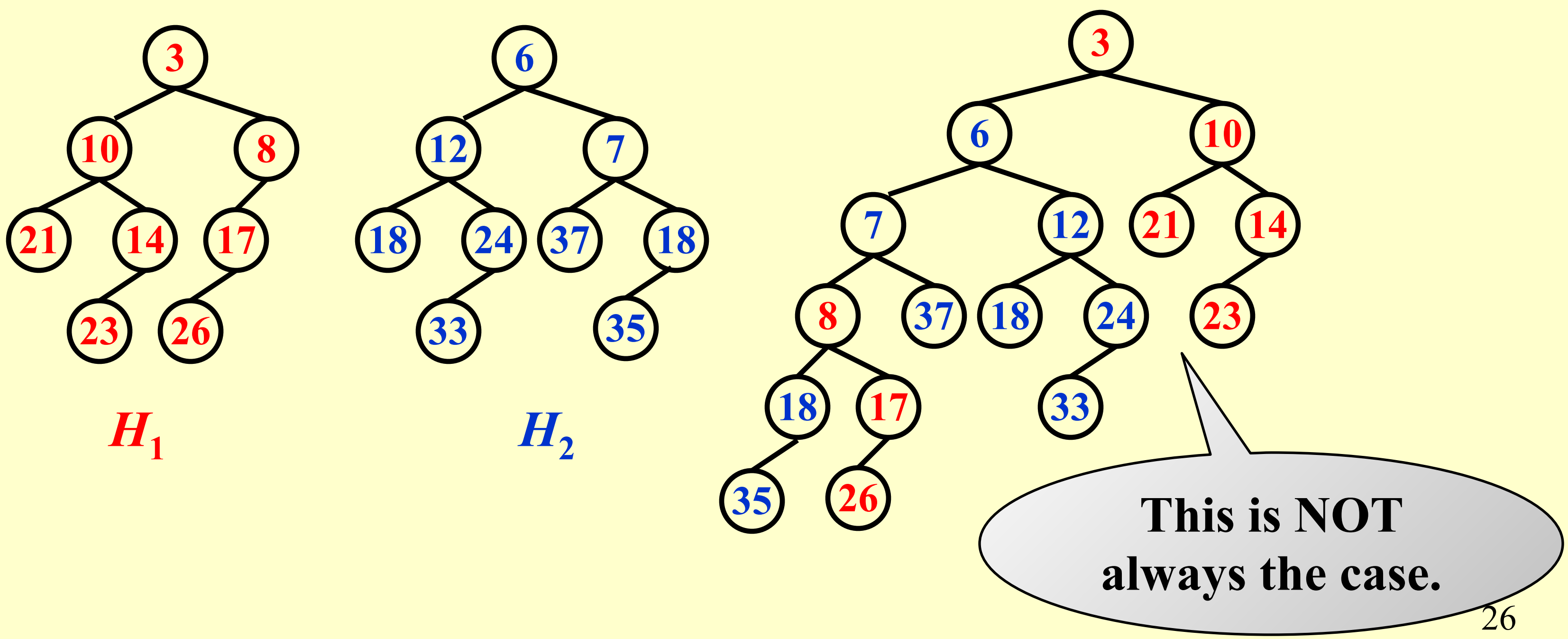
☞ **Merge: Always swap the left and right children except that the largest of all the nodes on the right paths does not have its children swapped. No Npl.**



Skew Heaps -- a simple version of the leftist heaps

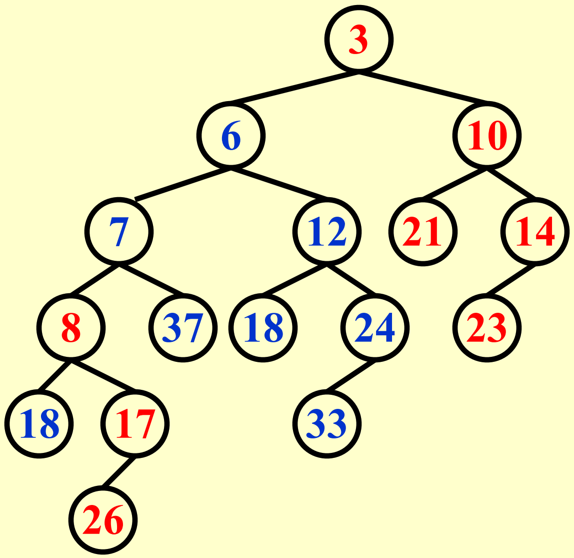
Target : Any M consecutive operations take at most $O(M \log N)$ time.

☞ **Merge:** Always swap the left and right children except that the largest of all the nodes on the right paths does not have its children swapped. No Npl.

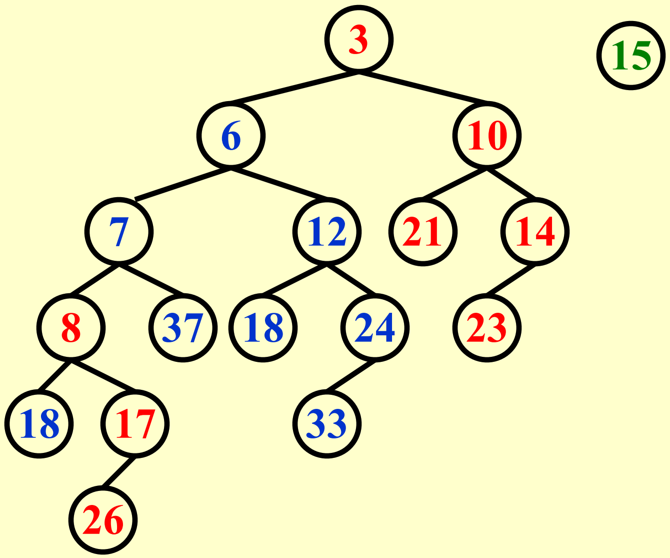


【Example】**Insert 15**

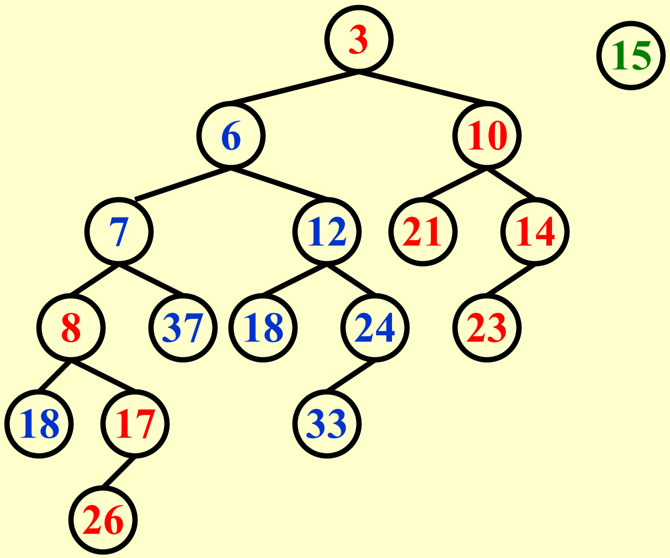
【Example】Insert 15

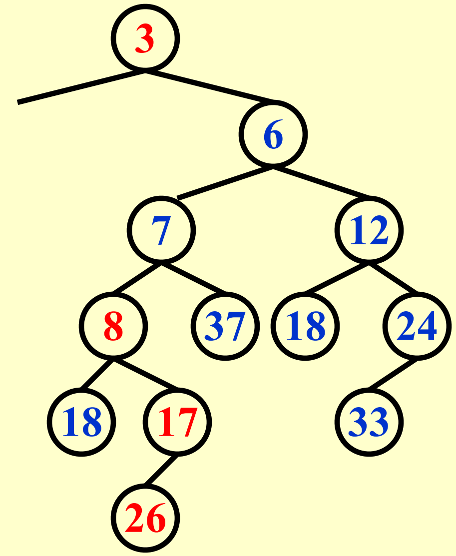


【Example】Insert 15

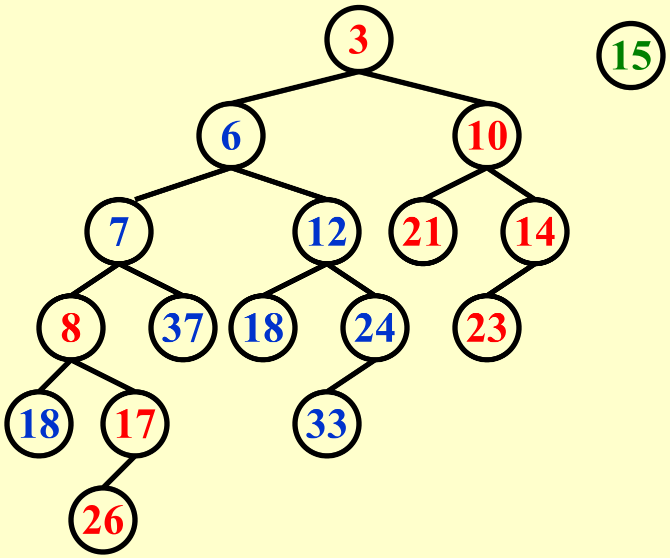


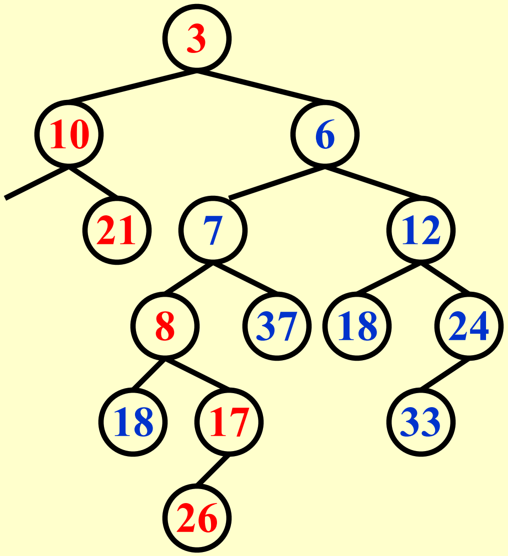
【Example】Insert 15



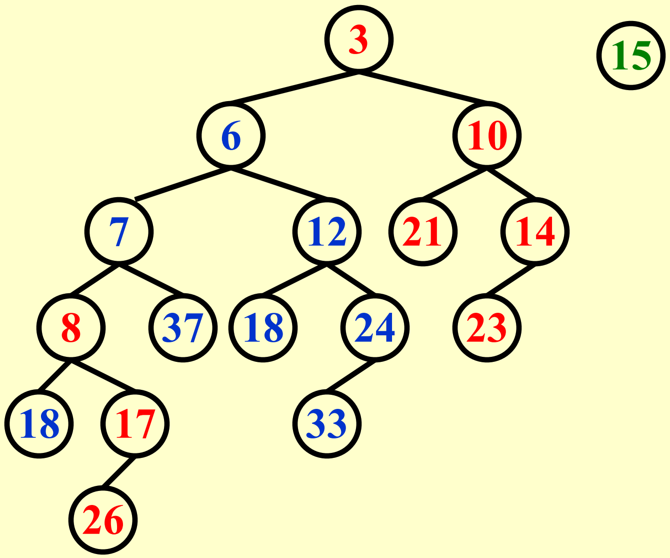


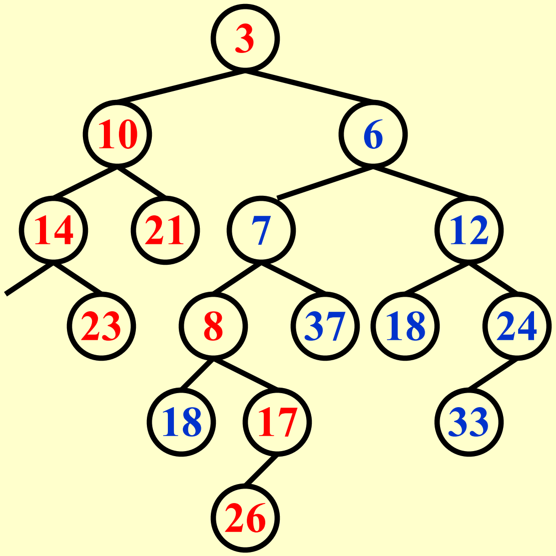
【Example】Insert 15



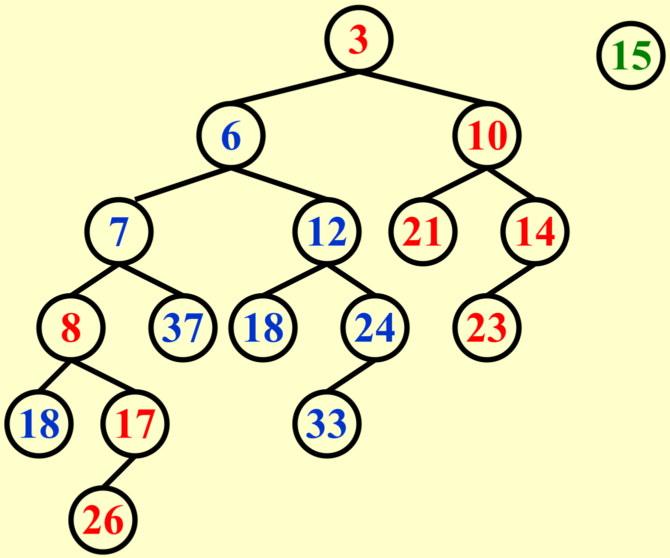


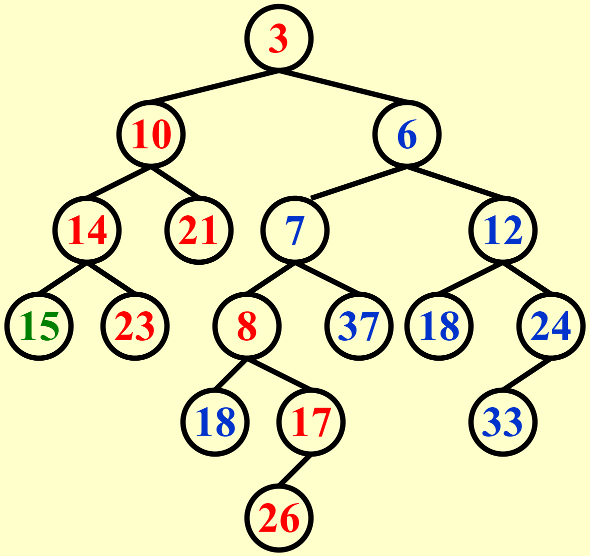
【Example】Insert 15



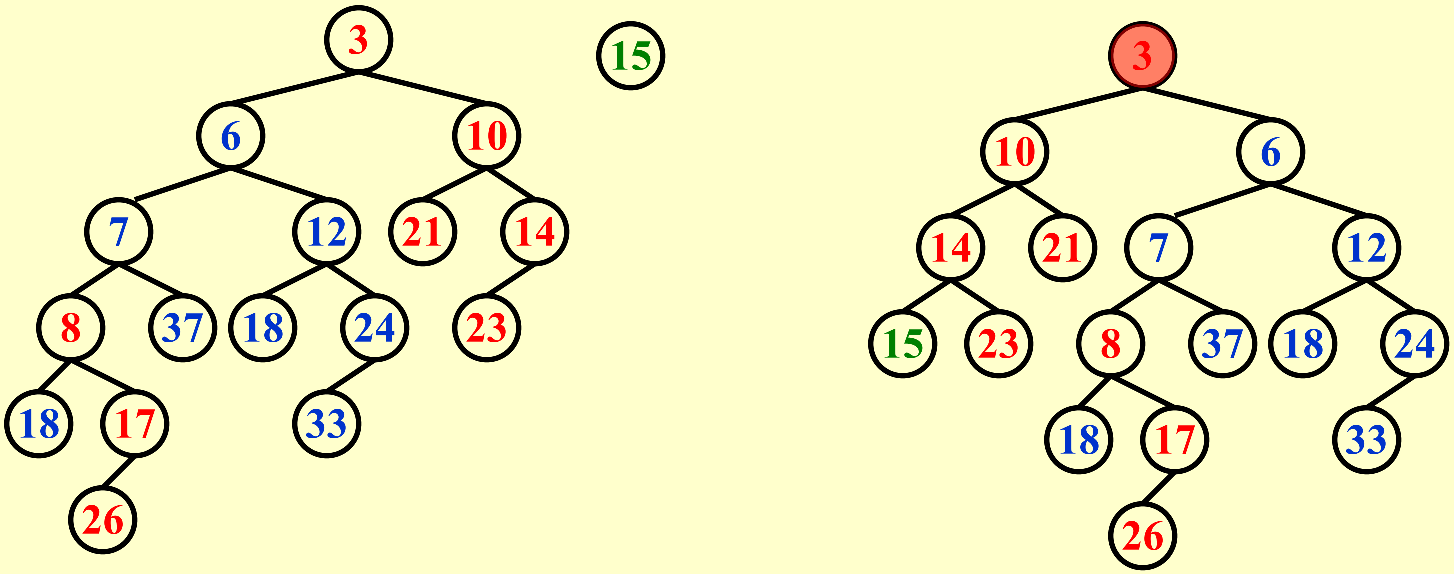


【Example】Insert 15

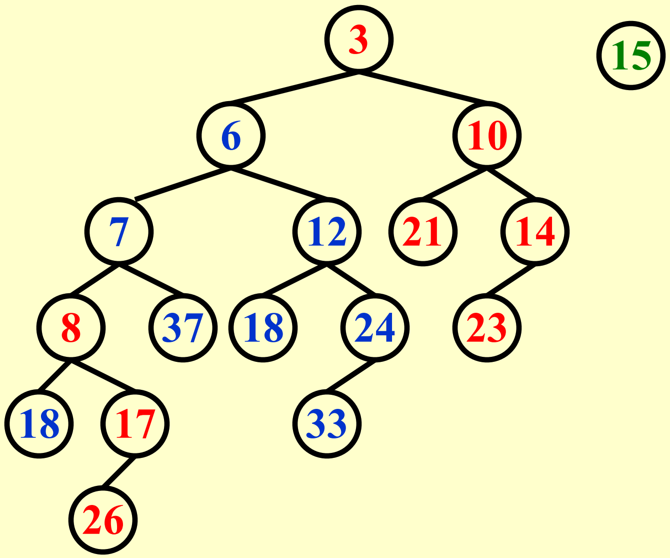


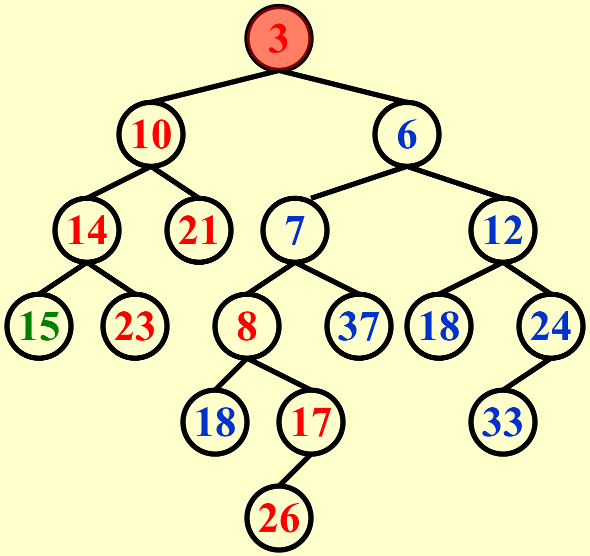


【Example】Insert 15



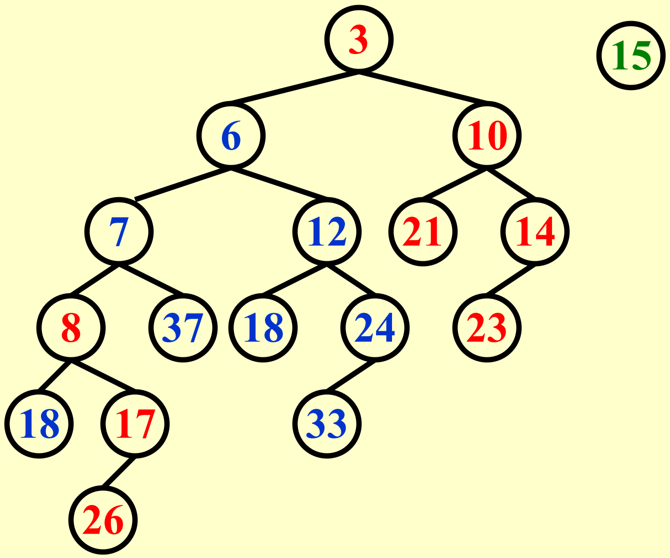
【Example】Insert 15

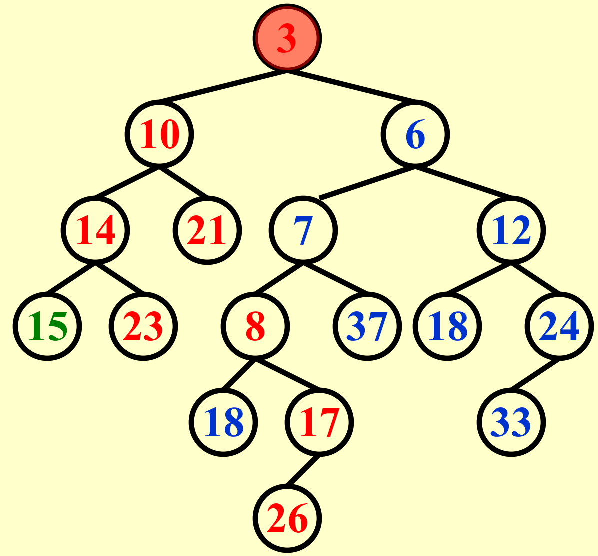




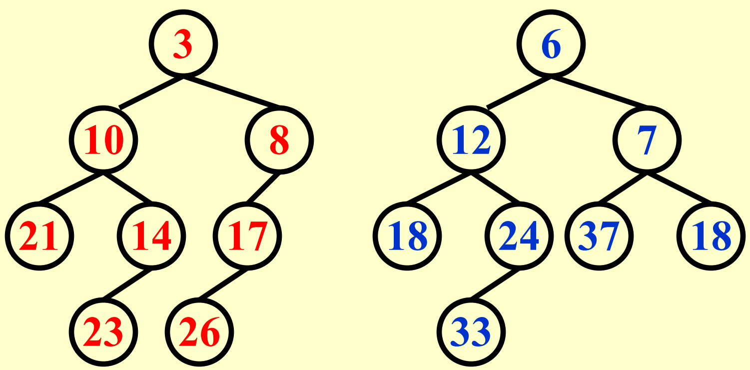
☞ **Merge (iterative version):**

【Example】 Insert 15



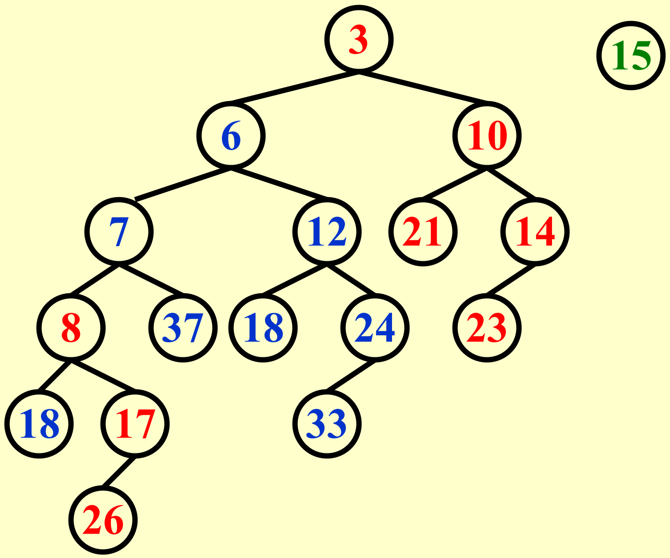


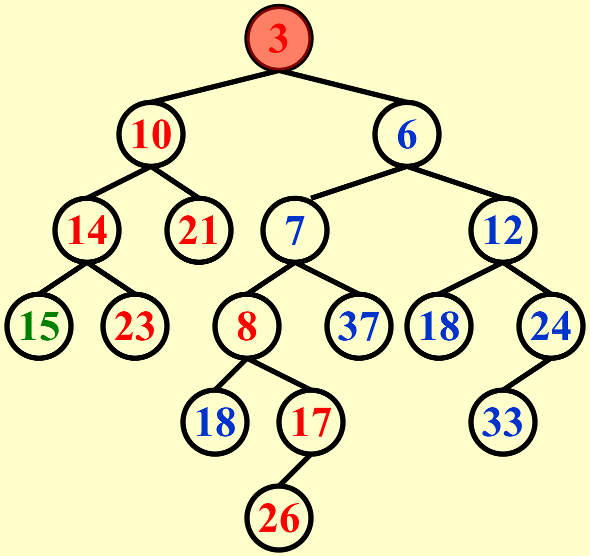
☞ **Merge (iterative version):**



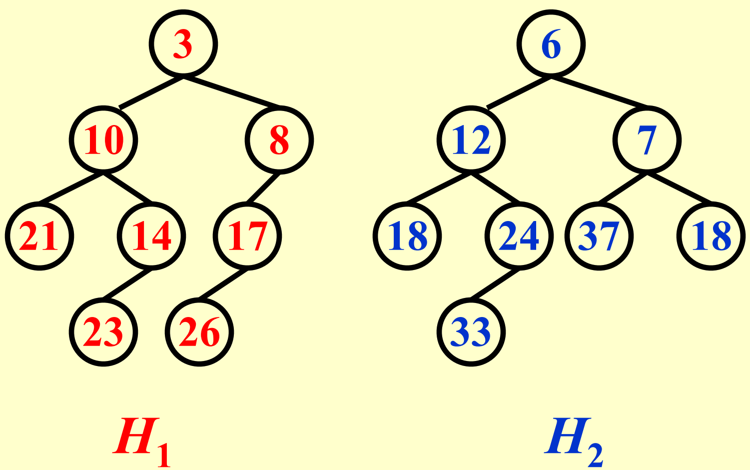
H_1 *H_2*

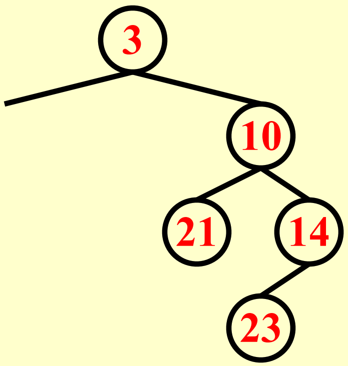
【Example】Insert 15



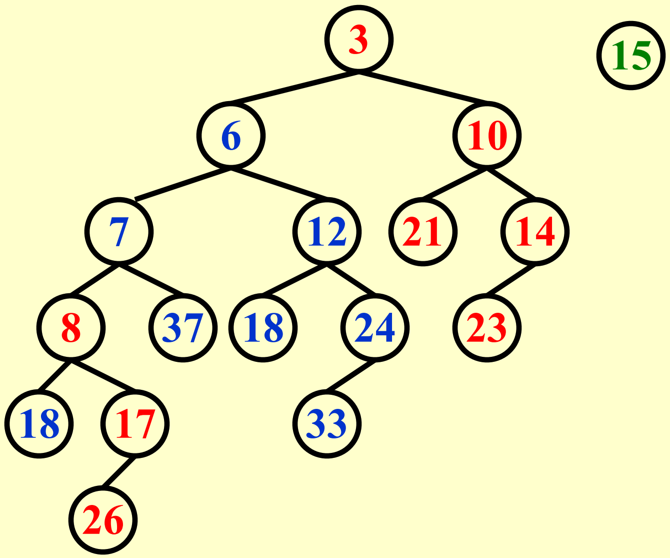


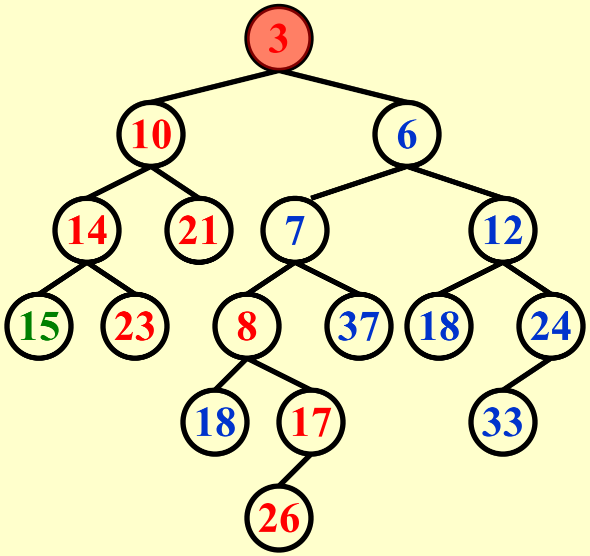
☞ **Merge (iterative version):**



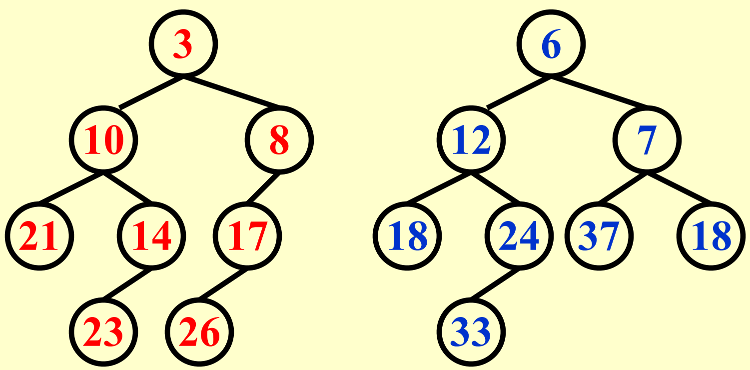


【Example】Insert 15

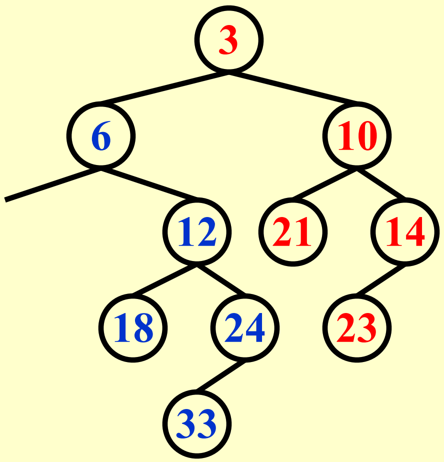




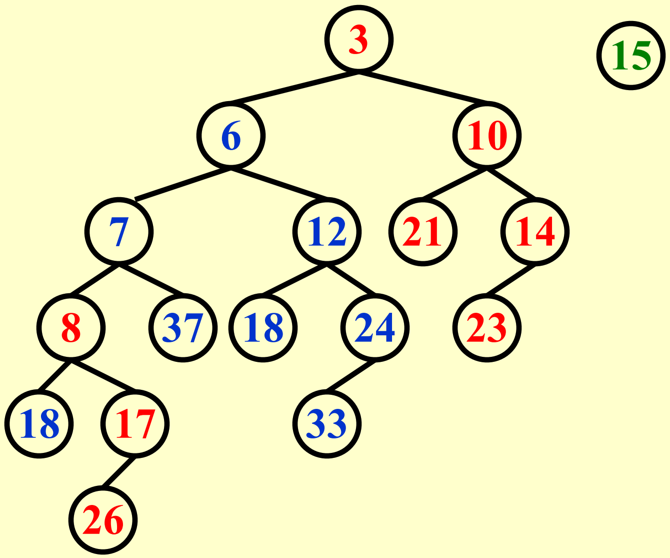
☞ **Merge (iterative version):**

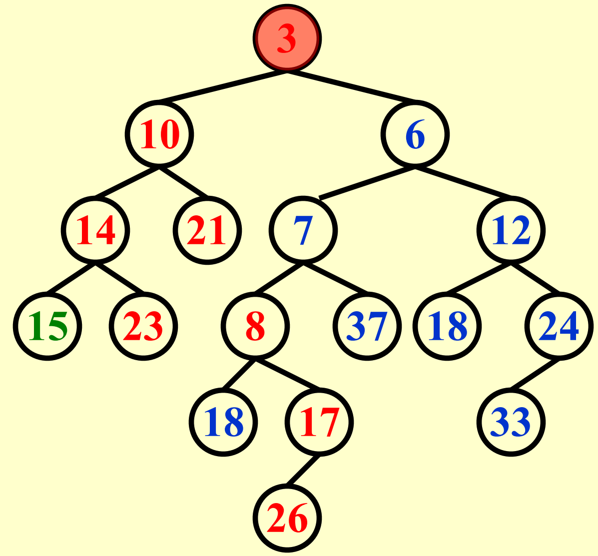


H$_1$ *H*$_2$

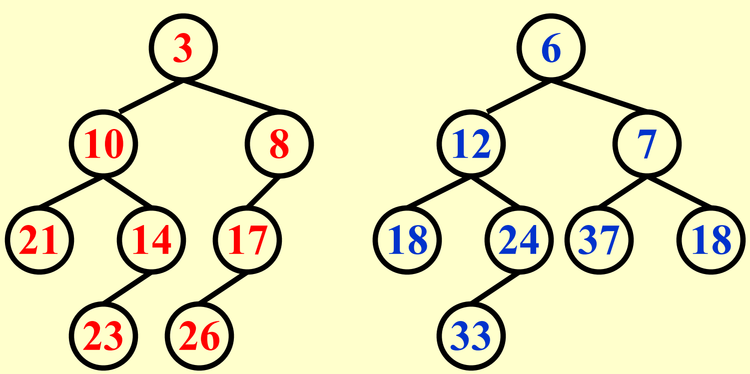


【Example】Insert 15

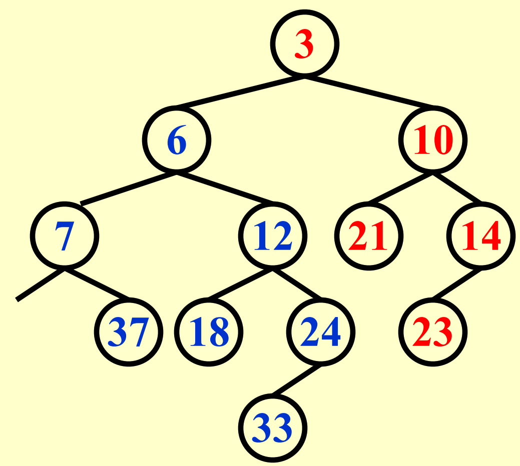




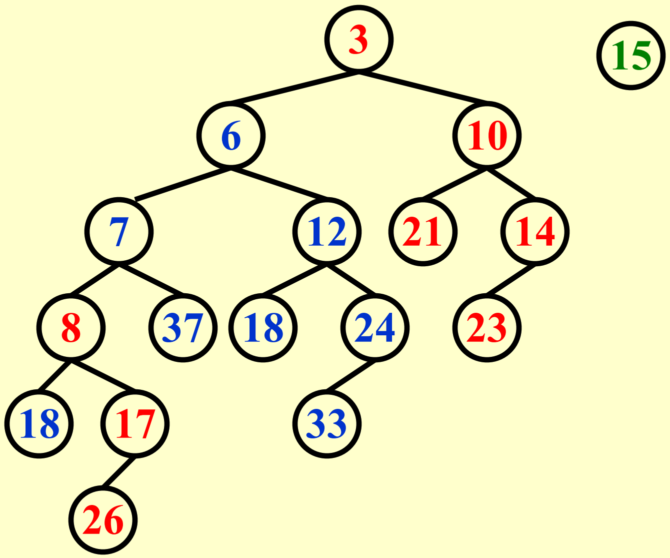
☞ **Merge (iterative version):**

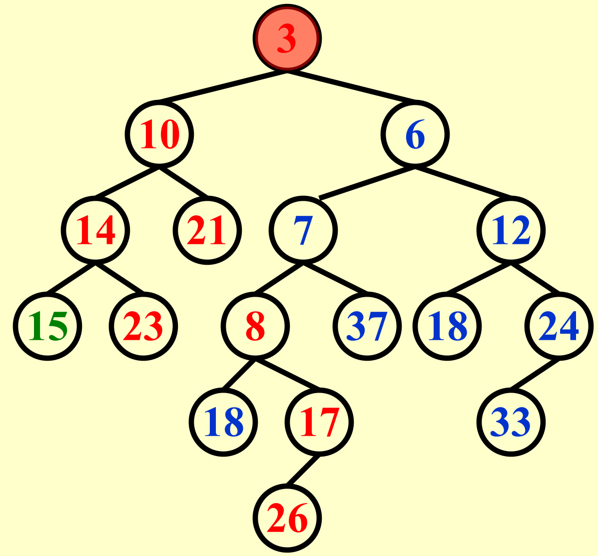


H_1 H_2

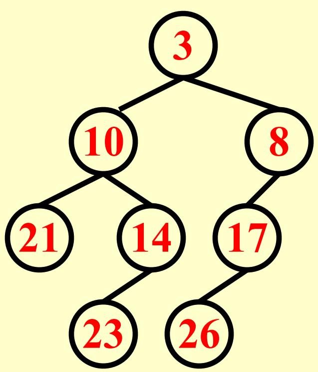


【Example】Insert 15

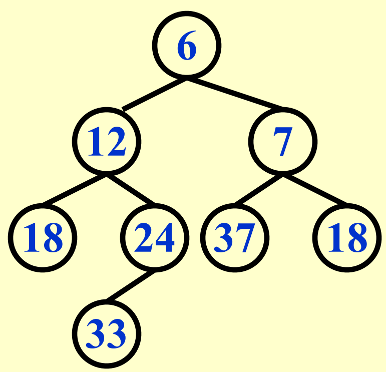




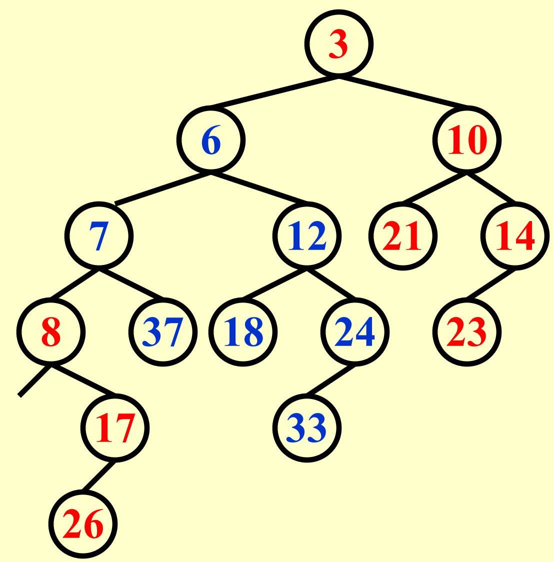
☞ **Merge (iterative version):**



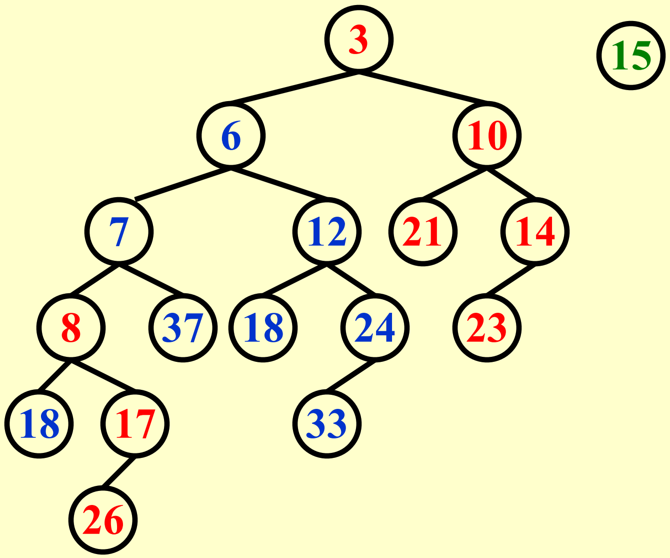
H_1

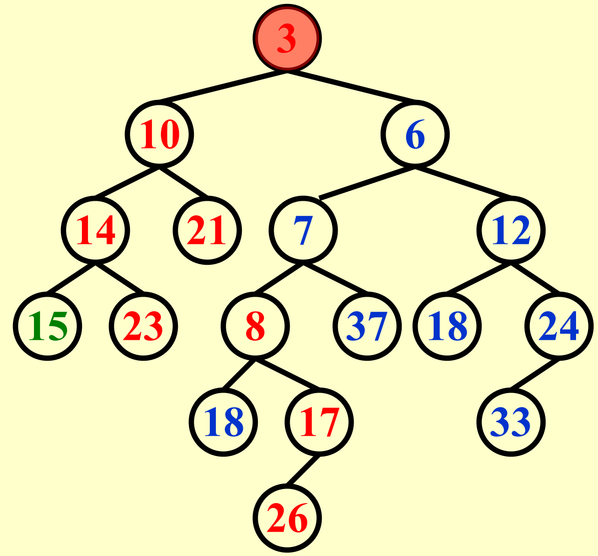


H_2

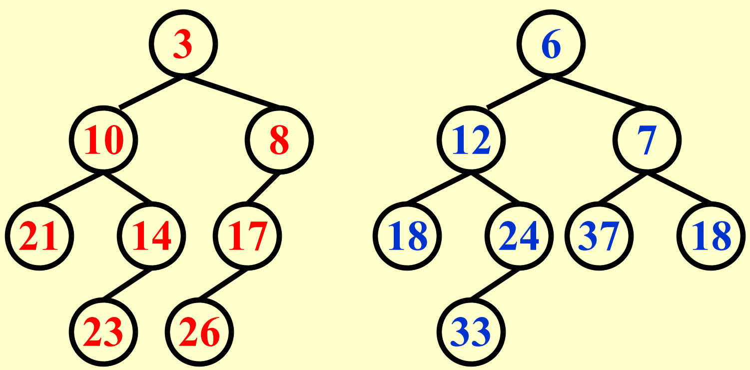


【Example】**Insert 15**

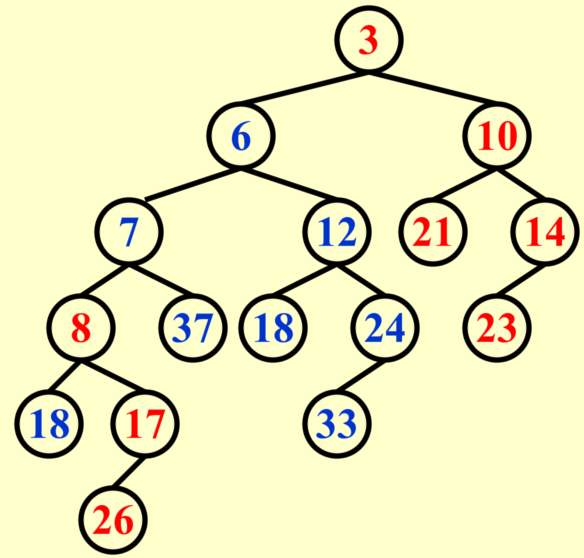




☞ **Merge (iterative version):**



H_1 H_2



Note:

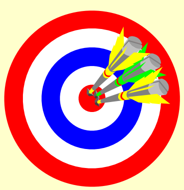
☞ **Skew heaps have the advantage that no extra space is required to maintain path lengths and no tests are required to determine when to swap children.**

☞ **It is an open problem to determine precisely the expected right path length of both leftist and skew heaps.**

Note:

☞ **Skew heaps have the advantage that no extra space is required to maintain path lengths and no tests are required to determine when to swap children.**

☞ **It is an open problem to determine precisely the expected right path length of both leftist and skew heaps.**



Target : **Any M consecutive operations take at most $O(M \log N)$ time.**

How to prove this?

Outline: Heaps (I)

- Review of Binary Heaps
- Leftist Heaps
- Skew Heaps
- Amortized analysis
- Take-home messages

Amortized Analysis for Skew Heaps

Amortized Analysis for Skew Heaps

Insert & Delete are just Merge

Amortized Analysis for Skew Heaps

Insert & Delete are just Merge

$T_{amortized} = O(\log N)$?

Amortized Analysis for Skew Heaps

Insert & Delete are just Merge

$T_{amortized} = O(\log N)$?

$D_i =$?

$\Phi(D_i) =$?

Amortized Analysis for Skew Heaps

Insert & Delete are just Merge

$T_{amortized} = O(\log N)$?

D_i = the root of the resulting tree

$\Phi(D_i) =$?

Amortized Analysis for Skew Heaps

Insert & Delete are just Merge

$T_{amortized} = O(\log N)$?

D_i = the root of the resulting tree

$\Phi(D_i) =$ number of right nodes?

Amortized Analysis for Skew Heaps

Insert & Delete are just Merge

$T_{amortized} = O(\log N)$?

D_i = the root of the resulting tree

$\Phi(D_i)$ = number of right nodes?

Amortized Analysis for Skew Heaps

Insert & Delete are just Merge

$T_{amortized} = O(\log N)$?

D_i = the root of the resulting tree

$\Phi(D_i)$ = number of ***heavy*** nodes

Amortized Analysis for Skew Heaps

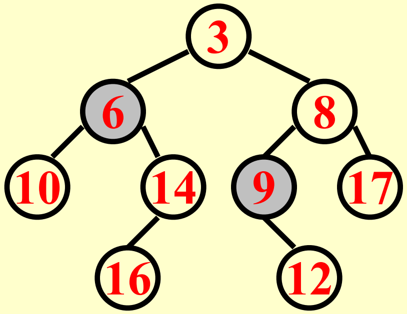
Insert & Delete are just Merge

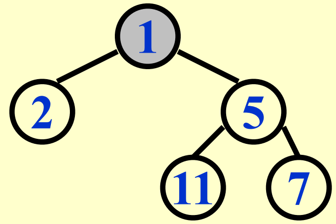
$T_{amortized} = O(\log N)$?

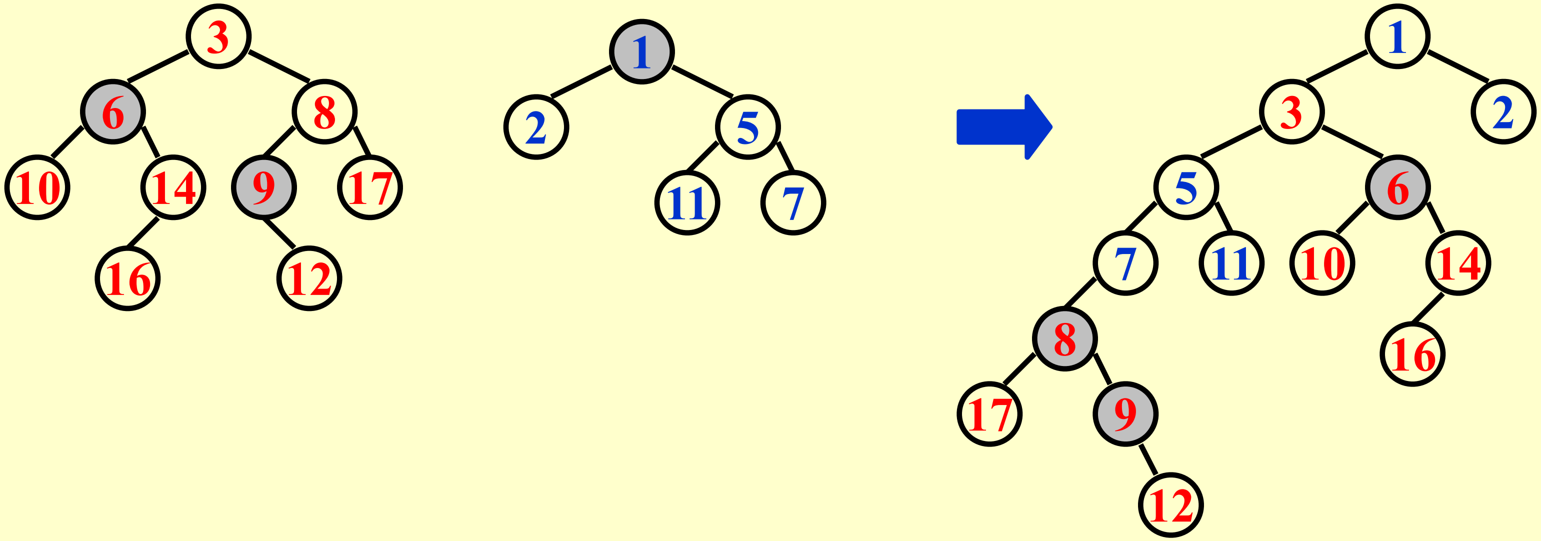
D_i = the root of the resulting tree

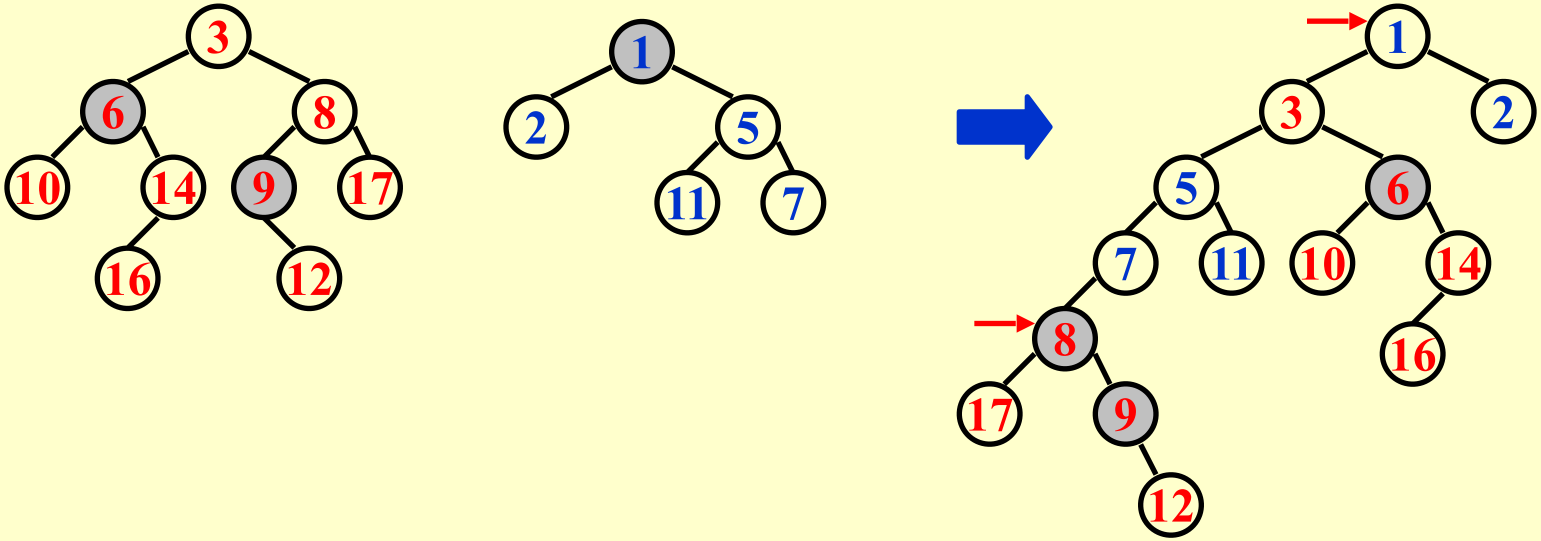
$\Phi(D_i)$ = number of ***heavy*** nodes

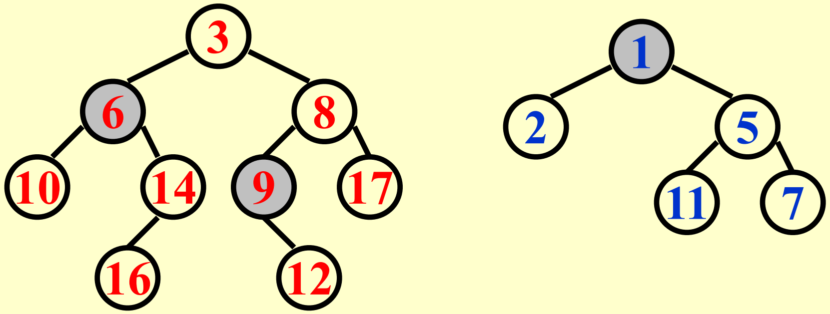
【Definition】A node *p* is *heavy* if the number of descendants of *p*'s right subtree is at least half of the number of descendants of *p*, and *light* otherwise. Note that the number of descendants of a node includes the node itself.

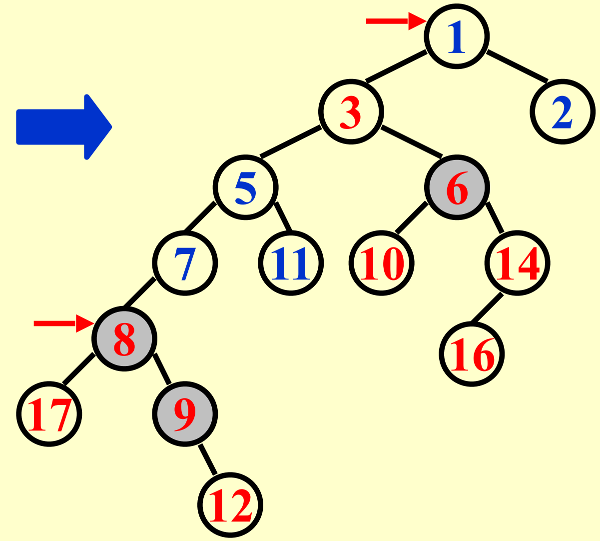

3
6
8
10
14
9
17
16
12


1
2
5
11
7

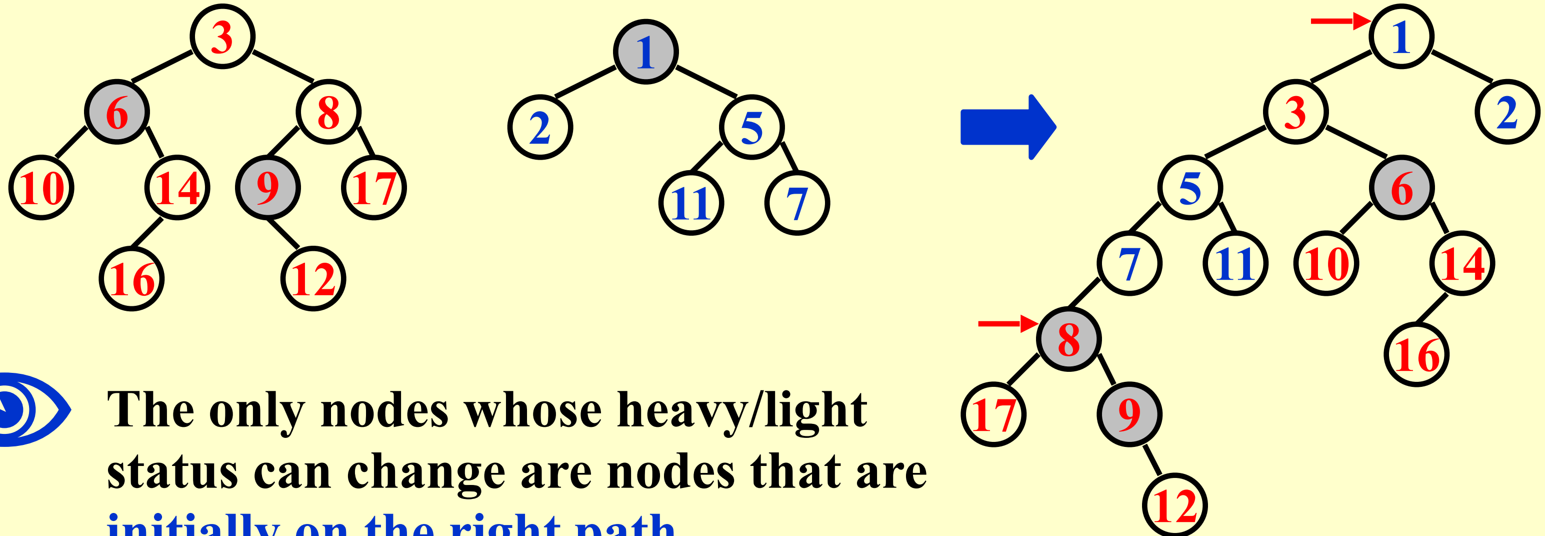

3
6
8
10
14
9
17
16
12
1
2
5
11
7
1
3
2
5
6
7
11
10
14
8
16
17
9
12


3
6
8
10
14
9
17
16
12
1
2
5
11
7
1
3
2
5
6
7
11
10
14
8
16
17
9
12



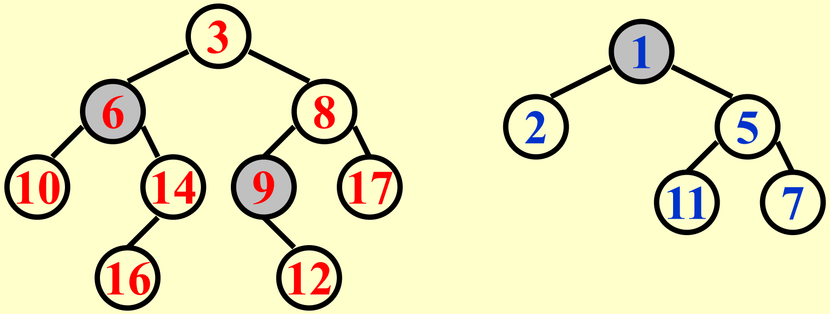


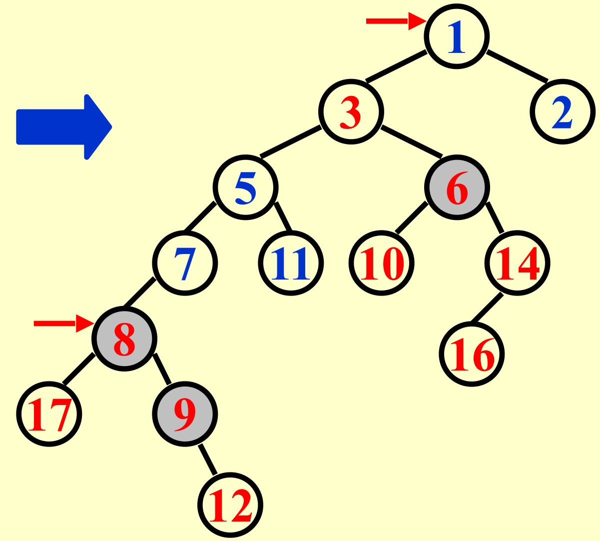
The only nodes whose heavy/light status can change are nodes that are initially on the right path.



The only nodes whose heavy/light status can change are nodes that are initially on the right path.

$H_i : l_i + h_i \;\; (i = 1, 2)$

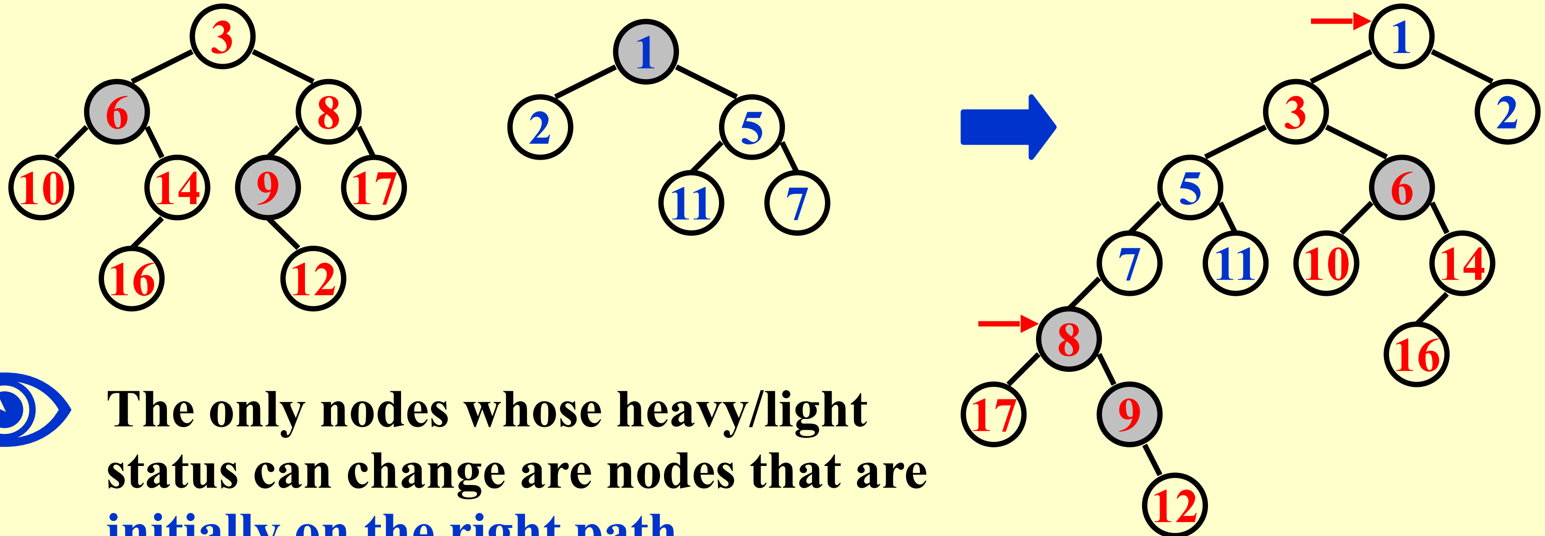




The only nodes whose heavy/light status can change are nodes that are initially on the right path.

$H_i : l_i + h_i \ \ (i = 1, 2)$

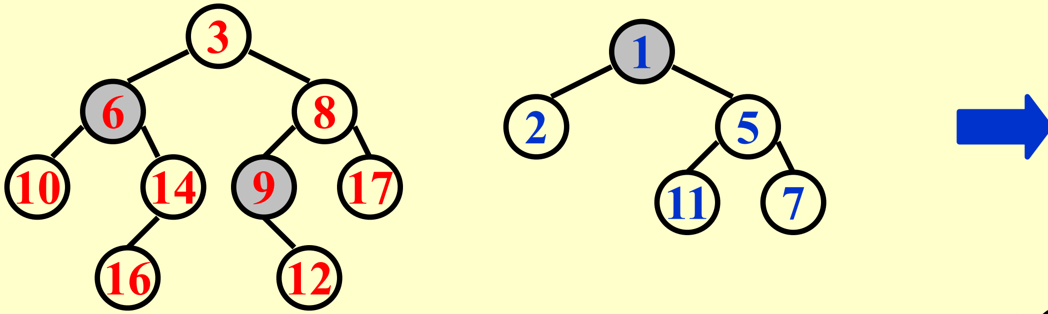
Along the right path

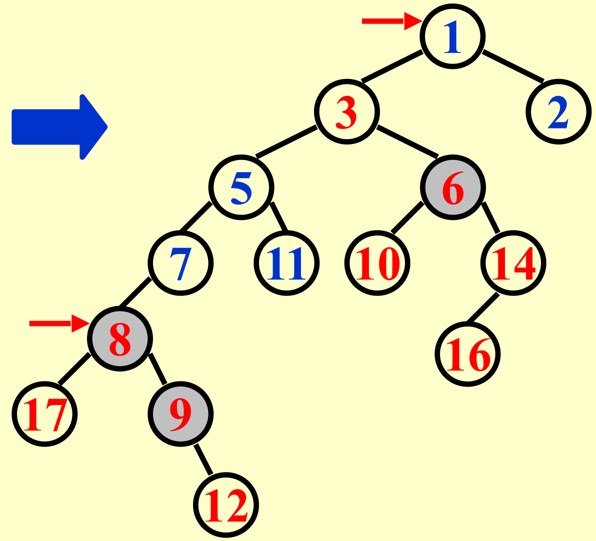


The only nodes whose heavy/light status can change are nodes that are initially on the right path.

$H_i : l_i + h_i \ \ (i = 1, 2)$ $\Rightarrow$ $T_{worst} = l_1 + h_1 + l_2 + h_2$

Along the right path





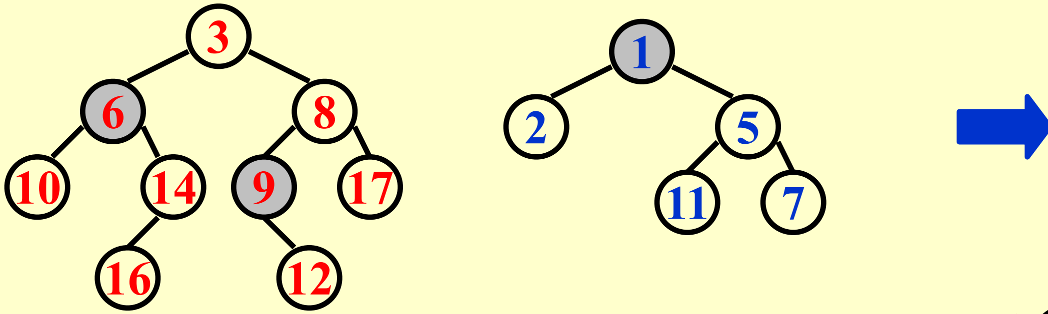
The only nodes whose heavy/light status can change are nodes that are initially on the right path.

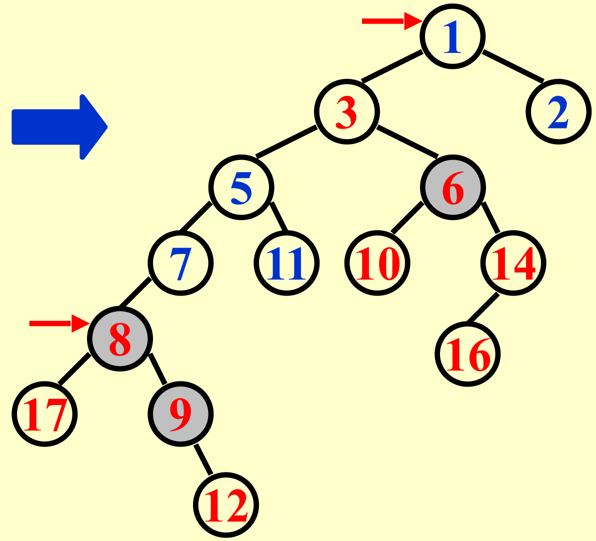
$H_i : l_i + h_i \ \ (i = 1, 2)$ ⟹ $T_{worst} = l_1 + h_1 + l_2 + h_2$

Along the right path

Before merge: $\Phi_i = h_1 + h_2 + h$

After merge: $\Phi_{i+1} \leq \ ?$





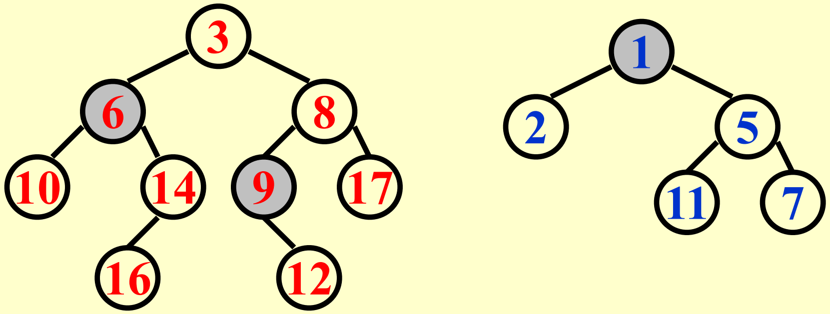
The only nodes whose heavy/light status can change are nodes that are initially on the right path.

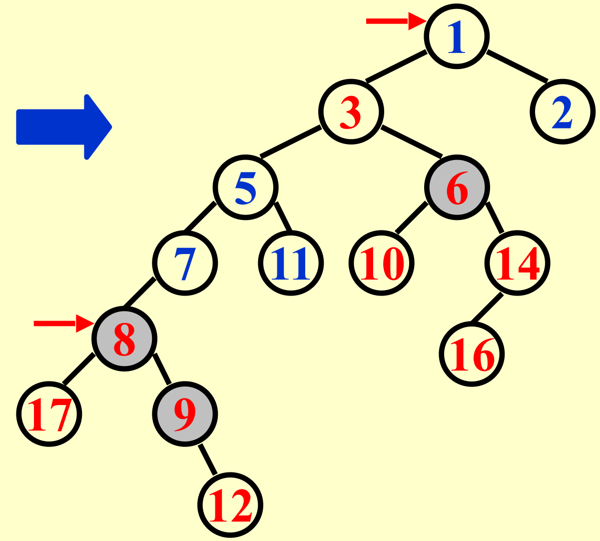
$H_i : l_i + h_i \ \ (i = 1, 2)$ ⇒ $T_{worst} = l_1 + h_1 + l_2 + h_2$

Along the right path

Before merge: $\Phi_i = h_1 + h_2 + h$

After merge: $\Phi_{i+1} \leq l_1 + l_2 + h$





The only nodes whose heavy/light status can change are nodes that are initially on the right path.

$H_i : l_i + h_i \;\; (i = 1, 2)$ $\Rightarrow$ $T_{worst} = l_1 + h_1 + l_2 + h_2$

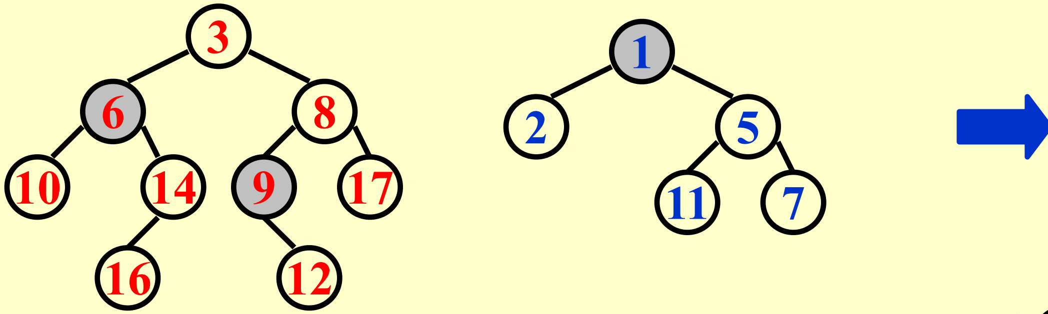
Along the right path

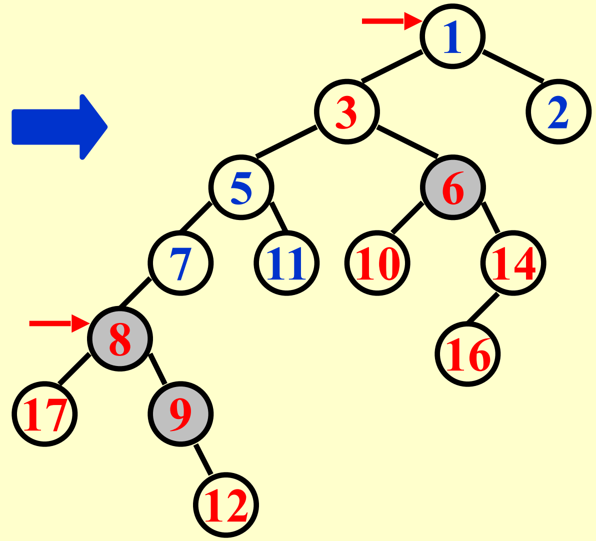
Before merge: $\Phi_i = h_1 + h_2 + h$

After merge: $\Phi_{i+1} \le l_1 + l_2 + h$

$$T_{amortized} = T_{worst} + \Phi_{i+1} - \Phi_i$$

$$\le 2\,(l_1 + l_2)$$





The only nodes whose heavy/light status can change are nodes that are initially on the right path.

$H_i : l_i + h_i \ \ (i = 1, 2)$ $\Rightarrow$ $T_{worst} = l_1 + h_1 + l_2 + h_2$

Along the right path

Before merge: $\Phi_i = h_1 + h_2 + h$

After merge: $\Phi_{i+1} \le l_1 + l_2 + h$

$T_{amortized} = T_{worst} + \Phi_{i+1} - \Phi_i$

$\le 2\,(l_1 + l_2)$

$l = O(\log N)$ $\Rightarrow$ $T_{amortized} = O(\log N)$

Outline: Heaps (I)

- Review of Binary Heaps
- Leftist Heaps
- Skew Heaps
- Amortized analysis
- Take-home messages

Take-Home Messages

- Leftist heaps:
  - Reduce merge cost to O(log N) by building unbalanced heaps, and push the computation on the right (light) paths.
- Skew heaps:
  - Avoiding skewness checking by always flipping left and right. Guarantee amortized cost O(log N).
- Amortized analysis:
  - The potential function measures how mess the data structure is.

Thanks for your attention!
Discussions?

Reference

Data Structure and Algorithm Analysis in C (2nd Edition)： Chap. 6.5-6.7, 11.3.

https://web.stanford.edu/class/cs166/lectures/06/Slides06.pdf

https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/heaps.pdf